Object-Oriented approach to code reuse.
Composition and Inheritance

Lecture 4

by Marina Barsky
Why use objects?

• Organization
  • Easier to change: the code is compartmentalized

• Encapsulation
  • Can be treated as a blackbox without knowing details

• Avoiding Repetition
  • Code reuse
Reusing objects

• We can build complex programs by reusing existing objects

• We can reuse code in two ways:
  • Composition
  • Inheritance
Reusing objects: composition
Objects as building blocks

• Instance variables can be of any type: they can also be of a new custom type (class)
• This way we can construct complex objects which contain simpler objects inside them
• The method of constructing a program by incorporating smaller objects inside a larger one is called **composition**
• This is **the most useful and widely used** approach in Object-Oriented Programming
Composing with objects

- While combining elementary objects we ensure that we expose only important properties and capabilities of these objects (contract, public interface)

- We can divide work among many programmers: each programmer can concentrate on correct implementation of each small piece

People who build engines do not have to know how to make wheels
Example: hospital

Hospital

- name: String
- patients: Patient []

- cureAll ()

Patient

- name: String
- age: int
- malady: String
- isCured: boolean

- cure ()

Contains
Start from a *Hospital* class – **pretend** that *Patient* class is already working

```java
public class Hospital {
    private String name;
    private Patient[] patients;
    int numPatients;
    int capacity;

    public Hospital(String name, int capacity) {
        this.name = name;
        patients = new Patient[capacity];
        this.capacity = capacity;
    }

    public void addPatient(Patient p) {
        if (this.numPatients < this.capacity)
            this.patients[this.numPatients++] = p;
        else
            System.out.println("...");
    }

    public void cureAll() {
        for(int i=0; i<numPatients; i++)
            patients[i].cure();
    }
}
```

Patient class is defined in a separate file, that can be written by another programmer
Define class *Patient*

```java
public class Patient {
    private String name;
    private int age;
    private String malady;

    public Patient(String name, int age, String malady) {
        this.name = name;
        this.age = age;
        this.malady = malady;
    }

    public Patient() {
        this.name = "John Doe";
        this.age = 25;
        this.malady = "unknown";
    }

    public void cure() {
        this.malady = "healthy";
    }
}
```

Default constructor— in case we don’t know
Running the Hospital

```java
public class RunHospital {
    public static void main (String [] args) {
        Hospital h = new Hospital("US Best", 10);
        h.addPatient(new Patient());
        h.addPatient(new Patient("Sally Smith", 21, "bruised ego"));
        h.addPatient(new Patient("Bob Swift", 18, "broken heart"));
        System.out.println("In the morning:");
        System.out.println(h);

        h.cureAll();
        System.out.println("In the evening:");
        System.out.println(h);
    }
}
```

The full code demo is [here](#)
Reusing objects: inheritance
Factoring-out similarities

- When we define a set of new types (classes) we often find that there are similarities among them.

- For example:
  - Class **Tiger** and class **Bear** – both have a lot in common: `move()`, `eat()`, `sleep()`, `makeNoise()`.
  - Instead of repeating these methods for each class, we can factor out similarities and define these methods in a single class **Animal**.
Where there’s inheritance, there’s an Inheritance Hierarchy of classes

- Mammal “is an” Animal
- Cat “is a” Mammal
- Transitive relationship: Cat “is an” Animal too

We can say:
- Reptile, Mammal and Fish “inherit from” Animal
- Dog, Cat, and Moose “inherit from” Mammal
Inheriting properties (fields) and capabilities (methods)

• Subclass *inherits* all capabilities of its superclass
  • if *Animals* eat and sleep, then *Reptiles, Mammals*, and *Fish* eat and sleep
  • if *Vehicles* move, then *SportsCars* move

• Subclass *specializes* its superclass
  • *adding* new fields and methods
  • *overriding* *(redefining)* existing methods

• Superclass *factors out* capabilities *common* among its subclasses
• Subclasses are defined by their *differences* from their superclass
Designing with inheritance

Superclass

Animal

name
diet
energyLevel
boundaries
location

makeNoise()
eat()
sleep()
move()

Subclasses

Lion
Cat
Tiger
Hippo
Dog
Wolf

package zoo;

public class Animal {
}

public class Bear extends Animal{
}

public class Cat extends Animal{
}
**Designing with inheritance**

**Instance variables** are the same for all animals.

So we define them all inside class Animal.

All the subclasses will inherit instance variables defined as **public** or **protected**.

---

**Access modifiers recap**

- **public**: accessible to all other classes
- **protected**: accessible to the class declaring it and its subclasses
- **no modifier**: accessible to the class declaring it and all classes in the same package
- **private**: accessible only to the class declaring it
Designing with inheritance

**Animal**
- name
- diet
- energyLevel
- boundaries
- location

**makeNoise()**
**eat()**
**sleep()**
**move()**

*sleep() and move() will be the same for all animals*

So we implement them only in the superclass, and each subclass has access to this code.
Designing with inheritance

Each animal will have their own `makeNoise()` and `eat()`
So we will override `makeNoise()` and `eat()`
defined in Animal() with the code specific to each subclass

<table>
<thead>
<tr>
<th>Animal</th>
<th>name</th>
<th>diet</th>
<th>energyLevel</th>
<th>boundaries</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>makeNoise()</td>
<td>eat()</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lion</th>
<th>makeNoise()</th>
<th>eat()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>makeNoise()</td>
<td>eat()</td>
</tr>
<tr>
<td>Tiger</td>
<td>makeNoise()</td>
<td>eat()</td>
</tr>
<tr>
<td>Hippo</td>
<td>makeNoise()</td>
<td>eat()</td>
</tr>
<tr>
<td>Dog</td>
<td>makeNoise()</td>
<td>eat()</td>
</tr>
<tr>
<td>Wolf</td>
<td>makeNoise()</td>
<td>eat()</td>
</tr>
</tbody>
</table>
Example of a superclass and a subclass

```java
public class Animal {
    protected String name;
    protected int energyLevel, x, y;
    protected String diet;

    public String getName() { return this.name; }

    public void move(int dX, int dY) {
        this.x += dX;
        this.y += dY;
        this.energyLevel -= (dX + dY);
    }

    public void eat() {
        System.out.println(name + " is eating " + diet);
        this.energyLevel ++;
    }

    public void sleep() {
        this.energyLevel ++;
    }

    public void makeNoise() {
    }
}

public class Cat extends Animal {
    public Cat() {
        super("Cat", "mice");
    }

    public void eat() {
        System.out.println("Cat is eating " + diet);
        this.energyLevel += 3;
    }

    public void makeNoise() {
        System.out.println("Purrr");
    }
}
```
Inheritance: constructor

- A subclass inherits all the members (fields, methods, and nested classes) from its superclass
- **Constructors are not inherited by subclasses**, but the constructor of the superclass can be invoked from the subclass

```java
public class Animal {
    public Animal() {
        this.name = "?";
        this.energyLevel = 100;
        this.x = 0;
        this.y = 0;
    }
    public Animal(String name) {
        this();
        this.name = name;
    }
    public Animal(String name, String diet) {
        this(name);
        this.diet = diet;
    }
}

public class Cat extends Animal {
    public Cat() {
        super("Cat", "mice");
    }
    ...
}
```
public class A {
    int iVar;

    public void hello() {
        System.out.println("Hello from A: " + iVar);
    }

    public void work() {
        iVar ++;
    }
}

public class B extends A {
    public void work() {
        iVar += 5;
    }
}

public class C extends A {
    public void hello() {
        System.out.println("Hello from C: " + iVar);
    }
}

public class B extends A {
    public void work() {
        iVar += 5;
    }
}

public class C extends A {
    public void hello() {
        System.out.println("Hello from C: " + iVar);
    }
}

IN MAIN:
A a = new A();
B b = new B();
C c = new C();
a.work();
b.work();
a.hello();
b.hello();
c.hello();

• A
  Hello from A: 1
  Hello from B: 5
  Hello from C: 0

• B
  Hello from A: 6
  Hello from B: 5
  Hello from C: 6

• C
  Hello from A: 1
  Hello from A: 5
  Hello from C: 0

• D
  None of the above
Polymorphism

• The reference and the object can be of different types in Java:

```java
Animal c = new Cat();
```

• We can treat the same object both as a subclass and as a superclass

• `c` can be used both as an `Animal` and as a `Cat`

• `c` has “many forms” – **polymorphism**

• We can use polymorphic variables as **method arguments**, **return types** or **array types**
Polymorphism: example

- Because *Dog*, *Cat* and *Lion* are also *Animals*, we can store them in array of *Animals*

- *makeNoise* is is declared in *Animal* (though it has an empty body), so we can call it on each element of the *Animal* array

```java
public class Animals {
    public static void main(String[] args) {
        Animal[] animals = new Animal[3];

        animals[0] = new Dog();
        animals[1] = new Cat();
        animals[2] = new Lion();

        for (Animal a: animals) {
            System.out.println(a);
            a.makeNoise();
        }
    }
}
```

Each animal makes their own noise
Why use inheritance

- Get rid of duplicate code by factoring out and implementing common behavior
- Modify in one place, and the change is ‘magically’ carried out to all subclasses
- Add new subclasses easily, and they have some methods and properties right away
- Guarantee that all classes grouped under a certain supertype have a common protocol
When to use inheritance

• When one class is a more specific version of another: 
  *SportsCar* extends *Car*

• When you have a method that is the same for a set of classes:
  *Square, Circle, Triangle* all need to have *move()* method in the animation program, so make *Shape* their superclass

• Test:
  • if you can say: *X IS A Y*, then use *inheritance*
  • If you can say: *X HAS A Y* use *composition*
“IS A” test

• Which of the following is the correct use of inheritance:
  A. class Oven extends Kitchen
  B. class Guitar extends Instrument
  C. class Ferrari extends Engine
  D. class Person extends Student
  E. None of the above
Java classes: single-root hierarchy

• All classes in Java (including our new custom classes) are subclasses of a single root superclass called *Object*

• When we create a new class that does not extend anything, this means *implicitly*:

```java
public class Dog extends Object
```

• This means that *Dog* inherits all the methods of *Object* (see [here](#))
So what’s in *Object*?

- Important public methods implemented in Object:
  
  ```java
  public String toString();
  public boolean equals(Object obj);
  public int hashCode();
  ```

- If you do not override these methods, you inherit them from the Object class
**toString()**

- *System.out* printing methods automatically call the `toString` method on their parameters
- By default, the `toString` method of an *Object* class returns a name of the new class and the memory location of the object
- If we do not override the `toString` method, then `toString()` of the nearest superclass will be used
public class Dog {
    private String name;
    private int height;

    public Dog(String name, int height) {
        this.name = name;
        this.height = height;
    }

    public static void main (String [] args){
        Dog d = new Dog("Fido", 15);
        System.out.println(d);
    }
}
Overriding default `toString()`

- We override the `toString` of `Object`
- We return a meaningful string representation of Dog’s state (instance variables)

```java
public class Dog {
    private String name;
    private int height;

    public String toString() {
        return "Here is Dog " + this.name + " " + this.height + " inches tall";
    }

    public static void main (String [] args) {
        Dog d = new Dog("Fido", 15);
        System.out.println(d);
    }
}
```

Here is Dog Fido 15 inches tall
equals ()

• In class Object, `o1.equals(Object o2)` returns true only if both `o1` and `o2` are references to the same place in memory – that is the default equals tests equality of references.

• We want to be able to compare objects themselves not their addresses.

• For this we override the default behavior of `equals()` according to the logic of our program.

• Note that `==` is still reserved for comparing references.
Comparing Dogs

public class Dog {
    private String name;
    private String diet="BONE";
    private int height;
    private String owner;

    public Dog(String name, int height) {
        this.name = name;
        this.height = height;
    }

    public static void main(String [] args) {
        Dog a = new Dog("Fido", 20);
        Dog b = new Dog("Ball", 10);
        Dog c = new Dog("Fido", 20);
        Dog d = a;

        System.out.println(a.equals(b));
        System.out.println(a.equals(c));
        System.out.println(a.equals(d));
    }
}
Comparing GoodDogs

```java
public class GoodDog {
    private String name;
    private String owner;
    private int height;

    public GoodDog(String name, int height, String owner) {
        this.name = name;
        this.height = height;
        this.owner = owner;
    }

    public boolean equals(GoodDog other) {
        return (this.name.equals(other.name)
                && this.owner.equals(other.owner));
    }

    public static void main(String [] args) {
        GoodDog a = new GoodDog("Fido", 20, "Sam");
        GoodDog b = new GoodDog("Fido", 20, "Bob");
        GoodDog c = new GoodDog("Fido", 20, "Sam");
        GoodDog d = a;

        System.out.println(a.equals(b));
        System.out.println(a.equals(c));
        System.out.println(a.equals(d));
    }
}
```

- What is printed?
  - A
    - false
    - false
    - false
  - B
    - false
    - true
    - true
  - C
    - false
    - false
    - true
  - D
    None of the above
How does Animal() look like?

• We factored out all the common code into class Animal

• However a generic Animal does not know how:
  
  `makeNoise()`
  `getPicture()`
  `getColor()`
  ...

• All these methods are not applicable to a generic class Animal

We want to prevent anyone from making an instance of Animal()

Animal class is too abstract!
Define Animal as abstract class

```java
public abstract class Animal {
    protected String name;
    protected int energyLevel;
    ...

    public void move(int dX, int dY) {
        this.x += dX;
        this.y += dY;
        this.energyLevel --;
    }

    public void eat() {
        ...
        this.energyLevel ++;
    }

    public void sleep() {
        this.energyLevel ++;
    }

    public abstract void makeNoise();

    public abstract Picture getPicture();
}
```

- Shared code which is applicable to all subclasses is still in concrete methods
- We can declare all the other methods `abstract`
- Abstract methods do not have body
- If the class has at least one abstract method, it must be declared abstract
- You must implement all abstract methods in a subclass
No instances of abstract animals

public abstract class Animal {
    protected String name;
    protected int energyLevel;
    ...

    public void move(int dX, int dY) {
        this.x += dX;
        this.y += dY;
        this.energyLevel --;
    }

    public void eat() {
        ...
        this.energyLevel ++;
    }

    public void sleep() {
        this.energyLevel ++;
    }

    public abstract void makeNoise();

    public abstract Picture getPicture();

    • You cannot create instances of an abstract class:
    Animal a = new Animal();
    This will not compile
Why use Abstract classes

• Inheritance allows to store shared code in a superclass
• Sometimes we cannot find any generic code useful to all subclasses
• In this case we declare a method in the superclass abstract (and the entire superclass becomes abstract)
• Even though there is no code in an abstract method, it still defines a common protocol that can be used in polymorphic programs: each subclass of Animal knows to makeNoise()
• Compiler forces all the subclasses to implement the abstract methods
Factoring out partial commonalities

- The Animal class defines a contract for all Lion, Hippo, Cat and Dog types
- We can use this hierarchy for Animal Simulation program
- But now I want to reuse some of the code for my Pet Store program
- I want to add `play()` method to some animals but not to all
- Basically I want some of the animals have an additional contract defined in superclass `Pet`
Java solution to multiple-inheritance problem

• Java does not allow a class to extend more than one superclass = it does not allow multiple inheritance

• However we can guarantee Pet behavior for all pet animals if we define all shared methods in a special Java class – Interface
  Not a GUI interface, not a colloquial use as in “public methods provide interface”, but a special Java keyword Interface
Pet interface

• In Interface all methods are abstract
• All subclasses must implement all of them
• Subclass extends a Superclass and implements Interface

```java
public interface Pet {
    public void play();
}

public class Dog extends Animal implements Pet {
    ...
    public void makeNoise() {
        System.out.println("Wuff");
    }

    public void play() {
        System.out.println("Dog playing");
        this.makeNoise();
    }
}

public class Cat extends Animal implements Pet {
```
Why use Interface

```java
public class PetStore {
    public static void main(String[] args) {
        Pet[] pets = new Pet[4];

        pets[0] = new Cat();
        pets[1] = new Cat();
        pets[2] = new Cat();
        pets[3] = new Dog();

        for (Pet p : pets) {
            p.play();
        }
    }
}
```

If all the methods in Interface are abstract – how is this the code reuse?

- A subclass can extend one superclass and implement multiple interfaces
- Common interface can be used for polymorphism
Which of the following is True?

A. You can't make an object of an Abstract class but you can of an Interface.

B. You can't make an object of an Interface but you can of an Abstract class.

C. You must implement all the abstract methods of the Interface, but you do not have to implement all the abstract methods of an Abstract class.

D. You can have both abstract and concrete methods in both Interface and Abstract class.

E. None of the above
Interface example: 1/3

Class Duck

<table>
<thead>
<tr>
<th>name</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

swim()
Interface example: 2/3

Class Duck
- name
- size

<<Interface>> Quackable
- quack()

<<Interface>> Flyable
- fly()

MallardDuck

RubberDuck

WoodenDuck

SpaceDuck
Interface example: 3/3

Class Duck
- name
- size

<<Interface>> Quackable
- quack()

<<Interface>> Flyable
- fly()

MallardDuck
- fly()
- quack()

RubberDuck
- quack()

WoodenDuck

SpaceDuck
- fly()