CSE 1325 - Object-Oriented Programming

Inheritance

Alex Dillhoff

University of Texas at Arlington

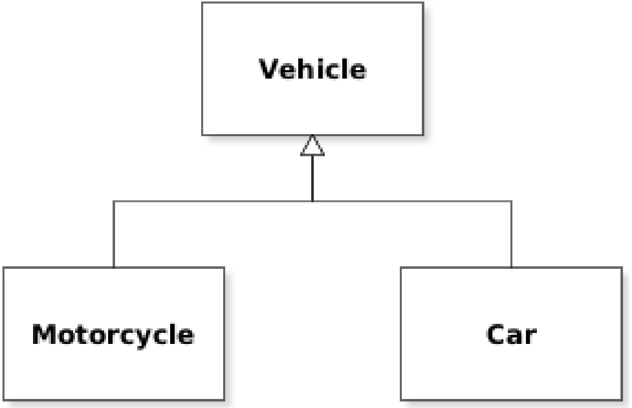
Inheritance in OOP

When designing software for a particular system, it is common to come across entities that share a subset of behaviors and properties.

Some of these relationships present themselves naturally. Others may be the result of a design decision.

Inheritance in OOP

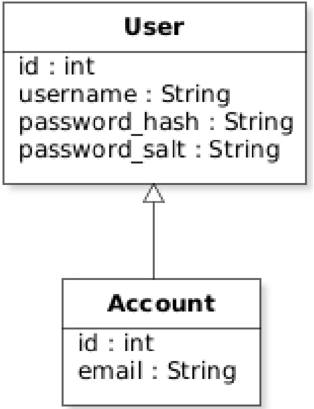
For example, a `Car` and `Motorcycle` are specific implementations of the broader category of `Vehicles`.



Inheritance in OOP

Inheritance based on a design decision would be creating a general `User` class which separates secure information like passwords from a particular `Account` class.

Inheritance in OOP



Account inherits from User.

Inheritance in OOP

The simple, yet inefficient, solution would be to copy the shared code between two classes.

The better solution is to create a class which `extends` the super class.

Inheritance in Java

The following code declares a new `Account` `class` that is a subclass of `User`.

```
public class Account extends User {
    ...
}
```

Inheritance in Java

The new class inherits the members and methods of the super class, but does not necessarily have direct access to them.

However, the superclass does not have access to any methods and fields defined in a subclass.

Inheritance in Java

As an example, let's apply inheritance to the RPG application.

What if we wanted to add a class to represent Non-Player Characters (NPCs) that the player's could face up against?

Inheritance in Java

This new class would need many of the attributes that are present in the `Player` class so that it could interact with the rules in the same way.

- `Name`
- `HP`
- `AC`
- `...`

Inheritance in Java

This class may also include fields and methods that are different from the `Player` class.

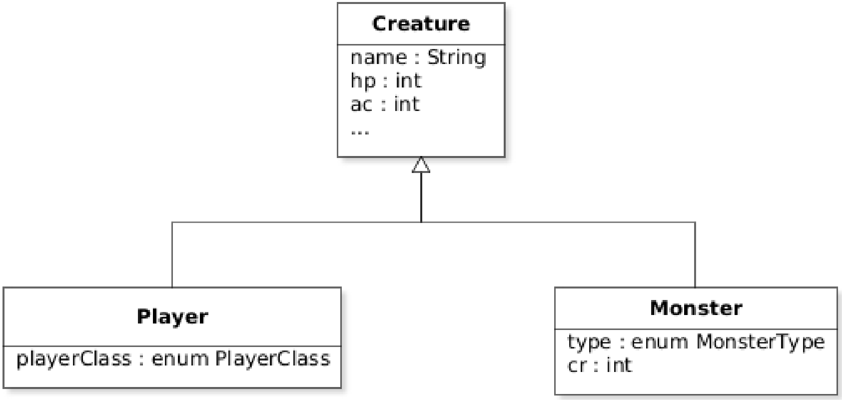
For example, we might want to include a challenge rating to inform the player's about how difficult a particular monster would be in combat.

Inheritance in Java

With inheritance, we can define a superclass named `Creature` that abstracts the shared attributes and methods.

We can then define `Player` and `Monster` as subclasses of `Creature`.

Inheritance in Java



Inheritance in Java

Example: Refactoring the `Player` class.

Overriding Methods

When creating subclasses, a common pattern calls for redefining methods.

In our RPG application, we want `Players` to benefit from bonuses differently than `Monsters`.

Overriding Methods

The Armor Class (AC) value will be calculated based on the `Player`'s Dexterity (DEX) value.

In order to implement this difference in the code, we will need to override the `getAc()` method in the `Player` class.

Overriding Methods

Since we do not have access to the private field ac, we need to call the superclass method getAc() when overriding this method.

```
public int getAc() {
    return super.getAc() + getDex();
}
```

Overriding Methods

When the method is being overridden, you must specify if you wish to call the superclass's method or the current class's method.

By default, the current class's method will be called first.

Working with Constructors

Another benefit for the subclass is that the superclass constructors can be reused without being redefined.

In our example, we set some default stat values for a the `Creature` superclass. It would be redundant to redefine those defaults in our subclass.

Working with Constructors

We can call on that superclass constructor quite easily.

```
public Monster(String name) {
    super(name); // Creature constructor
    cr = 1;
}
```

Working with Constructors

If no superclass constructor is specified, the no-argument constructor will be called automatically.

Polymorphism

Because `Player` and `Monster` both inherit from the same superclass, they can also be generally represented as `Creatures`.

This is especially useful when working with `Collections` like `ArrayList`.

Polymorphism

First, let's look at an example of how objects of these classes can be used in a general way.

Example: `PolymorphismTest.java`.

Polymorphism

In the previous example, we showed that a `Creature` object can refer to either a `Monster` or a `Player`.

This concept is referred to as **polymorphism**.

Polymorphism

We also observed that the correct `toString()` method was called during runtime.

This is enabled through a process called **dynamic binding**.

Dynamic Binding

How exactly does the compiler know which method to call?

Let's take the call from the previous example:

```
Creature c = creatures[0];
c.toString(); // Is it a Player or Monster?
```

Dynamic Binding

The virtual machine first matches the actual type of `c`.

If the actual type is a `Player`, then it will look in the class definition for a call to `toString()`.

Dynamic Binding

The virtual machine does not exhaustively search through the class definitions each time.

Instead, it creates a method lookup table that can be quickly referenced for such calls.

Dynamic Binding

If a definition of `toString()` does not exist in the `Player` table, it will search the superclass of `Player`.

This continues until it finds the method somewhere in the inheritance hierarchy.

Casting to a Subclass

We saw that a subclass can automatically be converted to a superclass reference.

What if we want to access a subclass's methods again?

We will need to cast it explicitly.

Casting to a Subclass

This can be done the same way we would cast any other type.

```
Monster m = (Monster) creatures[0];
```

Casting to a Subclass

However, if the underlying type of `creatures[0]` is **NOT** a `Monster`, a `ClassCastException` is thrown.

This exception either needs to be caught or we can use `instanceof`.

Casting to a Subclass

Example: Checking a cast with `instanceof`

Casting to a Subclass

This type of casting can only be done if the types are in the same inheritance hierarchy.

Most commonly, it will be used before casting from a superclass to a subclass.

Abstract Classes

A common situation that arises with class hierarchies is that of a superclass that does not have any specific implementations of methods or instances.

These classes are **abstract**.

Abstract Classes

Consider a program that draws different shapes.

A plausible class hierarchy would be one in which many specific shapes (triangle, circle, etc.) inherit from a general `Shape class`.

Abstract Classes

One method we might implement in this program is a `draw()` method that is overridden for each subclass of `Shape`.

It may be that the method in the `Shape` superclass does nothing at all!

Abstract Classes

Since `Shape` has no actual implementation, we can declare to be `abstract`.

```
public abstract class Shape {
    public abstract void draw();
}
```

Abstract Classes

Why not forego the abstract class and stick with the subclass implementations?

Example: Abstract Shape

Abstract Classes

It is not possible to create an instance of an abstract class.

```
Shape s = new Shape(); // Invalid
```

Abstract Classes

However, a variable of an abstract can be used to represent a subclass.

```
Triangle t = new Triangle();
Shape s = t;
```

Abstract Classes

Abstract classes can still have concrete instance fields and methods.

This is useful for superclasses that have attributes and methods shared by many subclasses.

Class Access

It is commonly considered good practice to make all instance fields `private`.

Access to the internal representation of an object should be facilitated via methods.

Class Access

There are, of course, exceptions that require different access.

Access to instance fields and methods can be restricted to classes within a hierarchy using the `protected` keyword.

Class Access

WARNING: Features that are marked as `protected` are also accessible within the entire package.

It is more common to mark methods as `protected` than it is to mark instance fields.

The `Object` Class

Every class `extends` the `Object` class in Java.

It is important to be familiar with a few features defined in this class.

The `Object` Class

One of the most important methods to override for custom classes is the `equals()` method.

Remember, comparing two objects using == only tests if they are identical references!

The `Object` Class

Example: Object Equality

The `Object` Class

When working with subclasses, the `equals()` method should first check the superclass `equals()` method.

```
// Returns false if `this` is not equal to `o`
if (!super.equals(o)) return false;
```

The Java Specification Rules of Equality

The official language specification describes some important properties that should be followed when overriding the `equals()` method.

The Java Specification Rules of Equality

Reflexivity

If `x` is not `null`, then `x.equals(x)` should return `true`.

The Java Specification Rules of Equality

Symmetry

For any `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.

The Java Specification Rules of Equality

Transitivity

For any `x`, `y`, and `z`, if `x.equals(y)` is `true` and `y.equals(z)` is `true`, then `x.equals(z)` should also be `true`.

The Java Specification Rules of Equality

Consistency

As long as the objects referred to by `x` and `y` remain unchanged, subsequent calls to `x.equals(y)` should also remain unchanged.

The Java Specification Rules of Equality

Null Equality

For a non-null `x`, `x.equals(null)` should return `false`.

The Object Class

Since every class extends `Object`, any class can be cast to `Object`.

```
Object myObject = new CustomClass();
```