Object-Oriented Programming

Inheritance

Inheritance

- In object-oriented programs, we use inheritance as one way to reuse program code.
- In Java, if class B **`extends`** class A, then B inherits (receives) all methods and fields from A.
  - Class B does not have to redefine these fields or methods.
  - Class A is called the superclass (or parent class).
  - Class B is called the subclass (or child class).

Example

Which class is the superclass and which class is the subclass?

`Vehicle`	**`Car`**
`Apple`	**`Fruit`**
`Square`	**`Rectangle`**

Inheritance (cont'd)

class B **`extends`** class A

- If the inherited variables or methods of A are **`public`**, these are accessible by instances of B (or users of these instances).
- If the inherited variables or methods of A are **`private`**, these are not directly accessible by instances of B (or users of these instances).

Inheritance (cont'd)

- In addition to the methods inherited by the superclass, the subclass can define its own fields and methods.
- These fields and methods are defined for the subclass but not for the superclass.

All classes are related

- Every class in Java inherits from another class, either explicitly (using **`extends`**) or implicitly.
- Example:

```
public class Taxi extends Car { ... }
```

- Classes that do not explicitly inherit from another class inherit from the Java class **`Object`**.
- Example:

```
public class Car extends Object { ... }
```

`Object`

- **`Object`** is the direct or indirect superclass of all classes in Java
  - except which one?
- Two methods inherited from **`Object`**:
  - **`public boolean equals(Object obj)`**
  - **`public String toString()`**
- Even if you don't write an **`equals`** or **`toString`** method for your class, your class has these methods since they are inherited from **`Object`**.

Inheriting from `Object`

- **`public boolean equals(Object obj)`**
  - Returns true if this object and the object in the parameter reference the same single object in computer memory.
- **`public String toString()`**
  - Returns a string that contains the name of the class followed by an @ symbol followed by the hexadecimal representation of the hash code of the object.

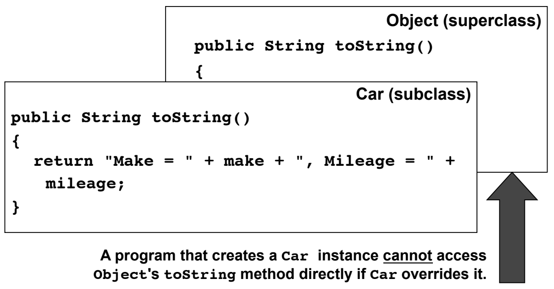
DO WE REALLY WANT TO INHERIT THESE?

Overriding methods

- A subclass can redefine inherited methods if the inherited method doesn't do exactly what the subclass needs.
- To override an inherited method, the subclass' method must use the exact same signature as the inherited method that is being overridden.
- If an inherited method is overridden, the user of the subclass cannot access the overridden method any longer.

Don't confuse overriding with overloading!

Overriding `toString`



Object (superclass)

```
public String toString()
{
```

Car (subclass)

```
public String toString()
{
  return "Make = " + make + ", Mileage = " +
   mileage;
}
```

A program that creates a `Car` instance <u>cannot</u> access `Object`'s `toString` method directly if `Car` overrides it.

What's wrong?

```
public String tostring()
{
  return "Make = " + make +
     ", Mileage = " + mileage;
}
```

Writing equals (the old way)

- Two cars are equal if and only if they have the same mileage and the same make.

```
public boolean equals(Car otherCar)
{
  return
      this.mileage == otherCar.mileage
      && this.make.equals(otherCar.make);
}
```

But this method doesn't override the inherited **`equals`** method from **`Object`** (not the same signature)!

Overriding equals (correctly)

- Override by using the same signature as in **`Object`**.

`equals` method in `Object` requires an `Object` parameter

```
public boolean equals(Object obj)
{
  Car otherCar = (Car)obj;
  return (this.mileage == otherCar.mileage
      && this.make.equals(otherCar.make);
}
```

Use typecasting to tell the compiler that the object really is a `Car`.

13

Overriding equals (incorrectly)

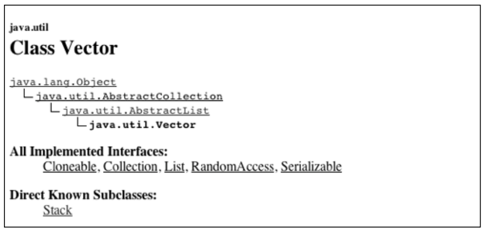
```
public boolean equals(Object obj)
{
  return (this.mileage == obj.mileage
          && this.make.equals(obj.make);
}
```

The `Object` class does not have a `mileage` or a `make` field.

14

Inheritance in the Java API

- Look at the Java API for the class **`Vector`**.



java.util

Class Vector

```
java.lang.Object
  └ java.util.AbstractCollection
      └ java.util.AbstractList
          └ java.util.Vector
```

All Implemented Interfaces:
Cloneable, Collection, List, RandomAccess, Serializable

Direct Known Subclasses:
Stack

15

What's an **`abstract`** class?

- An **`abstract`** class cannot be instantiated (constructed using a constructor).
  - It usually contains one or more abstract methods (methods that have a signature but no implementation).
  - Subclasses of abstract classes must provide an implementation for all inherited abstract methods by overriding the abstract methods.
- Example: Suppose an **`abstract`** class named **`Vehicle`** has **`Car`**, **`Truck`**, and **`Motorcycle`** as subclasses.
  - By defining the **`drive`** method as **`abstract`**, we leave it to the subclasses to define it, but all three classes must use the same signature (so all 3 vehicles drive "the same way").

16

Summary

- All classes in Java are related through inheritance.
  - We explicitly inherit from another class by using the keyword **`extends`** when we define the class.
  - We implicitly inherit from the class **`Object`** if we do not explicitly indicate a superclass.
- Although a class inherits from another class, we cannot access **`private`** variables or methods directly from the subclass.
- We can use the principle of <u>overriding</u> to redefine inherited methods.

17