COMP 250

Lecture 14

Inheritance 2:
visibility modifiers,
Object: hashCode, toString,
type conversion

Feb. 7, 2022
A subclass *inherits* the fields and methods of its superclass.

A subclass can also define its own fields and methods.
Inheritance relationships are not the same as package relationships.
e.g. a subclass can be in a different package than a superclass.

Let’s briefly go over some examples for how visibility modifiers
(public, private, ...) interact with inheritance relationships.
Inheritance and visibility : classes

For Class B to *extend* class A, it is necessary that class A is visible from class B. (Visibility was discussed in lecture 8.)

For example: two classes in the same package...

```java
package package1;
class A {
    :
}

package package1;
class B extends A{
    :
}
```
Inheritance and visibility: classes

For Class B to extend class A, it is necessary that class A is visible from class B. (Visibility was discussed in lecture 8.)

For example: two classes in different packages ...

```java
package package1;
public class A {
    :
}
```

```java
package package2;
import package1.A;
class B extends A{
    :
}
```
Inheritance and visibility: classes

For Class B to extend class A, it is necessary that class A is visible from class B. (Visibility was discussed in lecture 8.)

For example: two classes in different packages ...

```
package package1;

class A {
    :
}

package package2;

import package1.A;

class B extends A{
    :
}
```

Cannot import. Therefore cannot extend.

package ("package-private") visibility
Suppose class B extends class A (thus, class A is visible from class B). Which class A members are visible from class B?

“members” of a class A ≡ fields, methods, inner classes of A. “visible” here means that a class B method can name/refer to the class A member

The rules were given in lecture 8:

• **public** members of a class A are visible to class B.

• **private** members of a class A are *not visible* to class B.

• Package (called “package-private”) members of a class A are visible to class B only if classes A and B are in the same package.

See [here](#) for more details.
The class Beagle inherits the field/member name from class Dog.
package package1;

class Dog {
    private String name;
    :
}

package package1;

class Beagle extends Dog {
    Beagle(String name) {
        this.name = name;
    }
}

Example

Officially, the class Beagle does not inherit the field/member name from class Dog. (see lecture notes). But this just means that the name field isn’t visible. In fact, Beagle objects will have a name field.
The class Beagle inherits the field/member name from class Dog.
See similar case from two slides ago. The only difference now is that the classes are in two different packages.
package package1;
public class Dog {
    private String name;
    public void setName(String name) {
        this.name = name;
    }
}

package package2;
import package1.Dog;
class Beagle extends Dog {
    Beagle(String name) {
        this.setName(name);
    }
}

The Beagle object can set its name field using the inherited setName() method.
ASIDE: the protected modifier

It is less restrictive than the package ("package-private"). Use the protected modifier for a class member (field or method), if you want to allow the member to be visible to any class within the same package (like "package-private") and also to a subclass in another package.

```
package package1;
public class Dog{
    protected String name;
    : 
}

package package2;
import package1.Dog;
class Beagle extends Dog {
    Beagle(String name){
        this.name = name;
    }
}
```
COMP 250

Lecture 14

Inheritance 2:
visibility modifiers,
Object: hashCode, toString,
type conversion

Feb. 7, 2022
Last lecture: Object class

ASIDE:
the clone() method is protected and so we use a different symbol in the UML.

The reason clone() is protected are obscure and beyond scope of this course.
Object.hashCode()

Returns a (positive) integer. You can think of it as the address of the object, although this is not required in any technical sense.
Example: `hashCode()`

```java
class Object

+ equals( Object ) : boolean
# clone( ) : Object
+ hashCode( ) : int
+ toString( ) : String

```

```java
class String

+ equals( Object ) : boolean
# clone( ) : Object
+ hashCode( ) : int
+ toString( ) : String
```

String.hashCode() overrides Object.hashCode().

We will discuss String.hashCode()'s and other hashCode's in a few weeks, when we see how they are used.
Object.toString()

```java
class Object

+ equals( Object ) : boolean
+ clone( ) : Object
+ hashCode( ) : int
+ toString( ) : String
```

Example:

```java
Object obj = new Object();
System.out.println( obj );
```

Returned for me: `java.lang.Object@5305068a`
class Object

+ equals( Object ) : boolean
# clone( ) : Object
+ hashCode( ) : int
+ toString( ) : String

extends (automatic)

class String

+ equals( Object ) : boolean
# clone( ) : Object
+ hashCode( ) : int
+ toString( ) : String

returns className +
"@" +
Integer.toHexString( hashCode() )

String.toString() overrides Object.toString().

It returns the String object itself (not useful).
class Object

+ equals(Object) : boolean
# clone() : Object
+ hashcode() : int
+ toString() : String

extends (automatic)

returns className + "@" + Integer.toHexString(hashCode())

class Animal

+ toString() : String

Animal.toString() can override Object.toString(), if you wish. You can make it return whatever String you want. Typically, it contains info about that particular object.
See Hector’s discussion board posting for Assignment 1.

Course comp250 = new Course("COMP250", 3);

// add three students (Patty, Sam, Miranda) ....

System.out.println( comp250 );
Inheritance 2: visibility modifiers, Object: hashCode, toString, type conversion (or casting)

Feb. 7, 2022
Recall: Primitive Type Conversion (or Casting)

<table>
<thead>
<tr>
<th>Type</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
</tr>
<tr>
<td>char</td>
<td>2</td>
</tr>
<tr>
<td>byte</td>
<td>1</td>
</tr>
<tr>
<td>boolean</td>
<td>1</td>
</tr>
</tbody>
</table>

(wider) (does not require explicit cast)  

(narrower) (requires explicit cast)
Casting can be used for reference types also.

Heads up! Although a subclass is narrower, it actually has more fields and methods than the superclass (since it inherits all fields and methods from superclass).
Dog
- name: String
+ bark()

extends

Poodle
+ show()

extends

Beagle
+ hunt(): Animal
Dog myDog = new Poodle(); // upcast, widening

myDog.show(); X compile-time error!
Poodle has show() method, but Dog does not.
Dog    myDog = new Poodle();    // upcast, widening

Poodle  myPoodle = myDog;        compile-time error!

Implicit downcast Dog to Poodle is not allowed.
Dog  myDog = new Poodle();

Poodle  myPoodle = (Poodle) myDog;

myPoodle.show();

((Poodle) myDog).show();

What about runtime?
(next slide)
Dog myDog = new Poodle();

Poodle myPoodle = (Poodle) myDog;

myPoodle.show();

((Poodle) myDog).show();

object is indeed a Poodle
Poodle’s have a show method
We have given the compiler a heads up to expect myDog to reference a Poodle, and indeed the referenced object is a Poodle.

No Runtime errors either
Dog  myDog = new Beagle();

Poodle  myPoodle = (Poodle) myDog;  \[\text{Run-time error!}\]

myPoodle.show();

((Poodle) myDog).show();

\begin{itemize}
\item \textbf{Dog}  \\
- name: String \\
+ bark()
\item \textbf{Poodle}  \\
+ show()
\item \textbf{Beagle}  \\
+ hunt(): Animal
\end{itemize}
Sometimes we want to test at runtime whether an object is an instance of a specified class.

The `instanceof` operator can be used for this.

It returns `true` or `false`.

Dog d = new Dog();
System.out.println( d instanceof Dog);  // true

Beagle b = new Beagle();
System.out.println( b instanceof Dog);  // true

d = new Poodle();  // allowed
System.out.println( d instanceof Dog );  // true

System.out.println( d instanceof String);  // false
instanceof  and downcasting

We can use instanceof to make sure that downcasting will not cause a run time error.

class Test {

    static void dogMethod(Dog dog) {

        if (dog instanceof Beagle) {
            Beagle b = (Beagle) dog;
            b.hunt();
        }

        // :
    }

}
instanceof and equals()

We sometimes use instanceof when overriding equals():

```java
public class Shape {

    public boolean equals(Object obj) {

        if(obj instanceof Shape) {

            return this.getArea == ((Shape) obj).getArea();
        }

        else return false;
    }
}
```

FYI, the compiler does require that you have the modifier public and return type boolean since you are overriding the Object.equals(Object o) method. So the compiler does care about the signature.
Avoiding `instanceof`

It is tempting to use `instanceof` whenever you are unsure what the type of the object will be (at runtime), and you don’t want a runtime error.

But often it is unnecessary.

Let’s look at an example. We’ll say more next lecture.
Unnecessary use of `instanceof`...

```java
class Dog
    void bark()
        {print "woof"}

class Beagle
    void howl()
        {print "aowwwuuu"}

extends class Doberman
    void threaten()
        {print "Arh! Arh! Arh!"}

Dog dog;
    :
    if (dog instanceof Doberman)
        ((Doberman) dog).threaten();
    else if (dog instanceof Beagle)
        ((Beagle) dog).howl();
```
Instead... override
(example from last lecture)

```
class Beagle
void bark()
  {print "aowwwuuu"}

class Doberman
void bark()
  {print "Arh! Arh! Arh!"}
```

Dog myDog = new Beagle();
myDog.bark();

→ prints out “aowwwuuu”

myDog = new Doberman();
myDog.bark();

→ prints out “Arh! Arh! Arh!”

We will return to this example in an upcoming lecture.
ASIDE: Object.getClass()

An alternative way to compare objects is to use the Object class’es getClass() method.

For example,

dog1.getClass() == dog2.getClass()

How this works is an advanced topic and beyond the scope of the course. In particular, we won’t talk about the Class class.
## Coming up...

### Lectures

<table>
<thead>
<tr>
<th>Day</th>
<th>Date</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wed.</td>
<td>Feb. 9</td>
<td>Inheritance 3: (interfaces, abstract classes, polymorphism)</td>
</tr>
<tr>
<td>Fri.</td>
<td>Feb. 11</td>
<td>Inheritance 4: examples of interfaces: comparable, iterable</td>
</tr>
</tbody>
</table>

### Assessments

- **Assignment 1**
  - due on Friday, Feb. 11

- **Quiz 2** also on Friday, Feb. 11