

INFO1103: Introduction to Programming

School of Information Technologies, University of Sydney



Lecture 20: Inheritance

*Identifying similar types with different
behaviour, defining a general type, overriding
default behaviour*

Define a new type

A local shop sells a gourmet sandwich to its customers. The owner is very strict and only makes this perfected recipe for the fixed price.

The sandwich:

- has ingredients: bread, cucumber, tomato, anchovies, cornichons
- can be **assembled**, but has a specific order of 1) tomato, 2) anchovies, 3) cucumber, 4) cornichons
- costs exactly \$7.88

Create a new type called BestSandwich

Define another new type

Business is bad, the owner is convinced to progressively offer alternative sandwiches.

A chilli sandwich:

- has a name
- has ingredients: bread, chilli chicken, lime mayonnaise, cheese
- is served hot/cold
- requires **cooking** raw chicken.
- If serving cold, cooked chicken needs cooling.
- if serving hot, the mayonnaise is always added after other ingredients
- can be one of two sizes
- has a price \$4.90 or \$8.90 (size)

Create a new type called ChilliSandwich

Define many new types

Business improves. A few more kinds of sandwiches appear on the menu.

Each sandwich:

- has a price
- has a list of ingredients
- requires assembly



But, each sandwich:

- is assembled differently
- will have a different pricing based on the specific details
- will have different options for adding/removing ingredients
- will have different cooking processes involved

One class for **each** type of Sandwich because none of them are made the same

Subtype: “is-a” kind of

Types can be subtypes of other types.

For our sandwiches

- *BestSandwich* is a kind of *Sandwich*
- *ChilliSandwich* is a kind of *Sandwich*
-*Sandwich* is a kind of *Sandwich*

More examples

- *cat* is a kind of *pet*
- a *tiger* is a kind of cat
- an integer is a kind of number

“is-a” describes the relation from a **specific** type to a **general** type

Declaring a subtype: extends

New subtypes are defined using reserved word **extends**:

```
1 public class School {  
2     private String[] staff;  
3     private String[] students;  
4     // all my code for any kind of school  
5     public void gradeStudents() { ... }  
6 }
```

```
1 public class HighSchool extends School {  
2     // because I want to do some things differently  
3 }
```

```
1 public class University extends School {  
2     // because I want to do some things differently  
3 }
```

This means that my HighSchool will *inherit* all the methods that are not marked private in School, and it might have more methods and fields.

If one class 'A' *extends* another class 'B' we say A is a *subtype* of B.
Equivalently, if A "is-a" B then A is a *subtype* of B.

In this situation, B is also called a *supertype* of A.

When both A and B are classes, A is a subclass of B and B is a superclass of A.

E.g., *Pet* is a superclass of *DomesticDog* and *DomesticCat*.
DomesticCat is a subclass of *Pet*.

Apples and Oranges are Fruit

So Apple and Orange are subtypes of Fruit.

I can then write something like this:

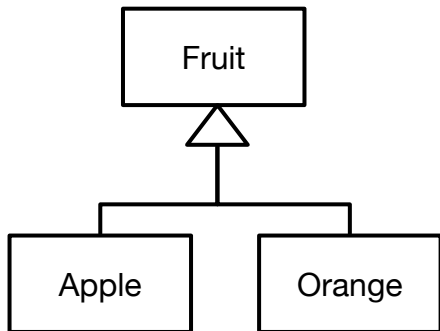
```
1 public class Fruit {
2     protected String name;
3     public void squeeze() {
4         // somehow extract the juice
5     }
6     protected boolean doesItBlend() {
7         // the most important question: will it blend?
8     }
9 }
```

```
1 public class Apple extends Fruit {
2     /* Inherits all the methods and variables not marked
3     private in Fruit
4     May have additional methods and variables.
5     */
6 }
```

Apples and Oranges are Fruit (cont.)

(Similarly for Orange class).

Now my inheritance diagram looks a little more complex:



Access Modifiers matter here

Now that we are considering inheritance, it's clear that the access modifiers matter more.

Use **private** to mark methods/fields as only directly accessible from within the class

Use **protected** to allow access to subclasses or other classes

Use **public** to allow global access

Use the default (**no modifier**) to permit access to any class in the same package (but not subclasses)


Casting among Classes

Remember we can cast a char as an int, and an int as a float?

```
1  int i = 3;
2  float f = 2.1f;
3  i = (int) f;
4  float x = (float) i;
5  char ch = 'c';
6  int y = (int) ch;
```

We are allowed to cast a variable 'a', which is of type "A" to be treated like another type, say "B", like this:

```
1  Apple a = new Apple();
2  Fruit f = (Fruit) a;
```

 When will casting a different class not work?

Casting among Classes (cont.)

```
1 Apple a = new Apple();
2 Orange o = new Orange();
3
4 Fruit firstFruit = (Fruit) a; // this is fine
5                               // the dynamic type of a is
6                               // Apple, which is a subtype (and subclass)
7                               // of Fruit
8
9 Fruit secondFruit = (Fruit) o; // Also fine
10                              // the dynamic type of o is
11                              // Orange, which is a subtype of Fruit
12
13 Apple b = (Apple) o; // Won't compile: Apple is not a subtype
14                     // of Orange
15
16 Elephant e = (Elephant) a; // don't be silly
```

The *subtype* of a thing by default gets all the behaviour that the *supertype* has.

That means methods available in the superclass that are not marked “private” or otherwise made inaccessible can be used by *subtypes*.

Let's see how this works for a simple example.

Inheritance example

```
1 public class Foo {
2     /* only objects in Foo class can see this
3        if I marked it protected, all subclasses could access
4        it too, as well as all classes in this package */
5     private String name;
6
7     public Foo(String s) { // constructor
8         name = s;    // set the name to the argument s
9     }
10    public String greet() { // return a friendly message
11        return "Hello, I am a Foo! and my name is " + name;
12    }
13    public String toString() {
14        // overwrites the Object toString
15        return greet();
16    }
17 }
```

Inheritance example (cont.)

```
1 public class Bar extends Foo
2 {
3     // constructor
4     public Bar() {
5         super("I have no name. :(");
6         // "super" by itself calls the constructor of the
7         // superclass
8     }
9
10    public Bar(String s) {
11        super(s);
12        // call the Foo constructor that takes a String
13    }
14
15    @Override
16    // This indicates greet() is overriding a method in Foo.
17    // Optional, but the compiler will check if it's present.
18    public String greet() {
19        return "Hello, I am a Bar, which is a kind of Foo!";
20    }
21 }
```


Inheritance example (cont.)

```
1 public class BarAsFooTest {
2     public static void main(String[] args)
3     {
4         Foo foo = new Foo("Harry"); // make a new Foo
5         Foo bar = new Bar(); // static and dynamic types differ
6         Foo c = new Bar("James"); // assign a name to the Bar
7
8         Bar x = new Foo("Xavier"); // type cast a Foo as a Bar
9         Bar y = (Bar)foo; // type cast a Foo as a Bar
10
11         System.out.println(foo.greet());
12         System.out.println(bar.greet());
13         System.out.println(c.greet());
14         System.out.println(x.greet());
```

Does this compile?

What does this print?

Running the Foo/Bar program:

```
Hello, I am a Foo! and my name is Harry  
Hello, I am a Bar, which is a kind of Foo!  
Hello, I am a Bar, which is a kind of Foo!
```

All Bars are Foos

If all Bar objects are Foo objects, then we could treat a collection of Foo and Bar objects all as Foos. Right?

All Bars are Foos

If all Bar objects are Foo objects, then we could treat a collection of Foo and Bar objects all as Foos. Right?

Right!

```
3 {
4     Foo foo = new Foo("Harry"); // make a new Foo
5     Foo bar = new Bar(); // static and dynamic types differ
6     Foo c = new Bar("James"); // assign a name to the Bar
7
8     Foo[] fooArray = new Foo[3];
9     fooArray[0] = foo;
10    fooArray[1] = bar;
11    fooArray[2] = c;
12    // apply idiom to do something with all Foo types
13    for (int i = 0; i < fooArray.length; i++) {
14        System.out.println( fooArray[i].greet() );
15    }
16 }
```

Running the Foo/Bar program again:

```
Hello, I am a Foo! and my name is Harry  
Hello, I am a Bar, which is a kind of Foo!  
Hello, I am a Bar, which is a kind of Foo!
```

Infrequently Asked Question:

Is it possible to have fields in a subclass with the same name as those in the superclass?

Duplicate fields in related classes



It is possible, yes, but it's a Terrible Idea.

```
1 public class Foo {
2     protected int x;
3     // x in the superclass
4     private String label;
5
6     public Foo(String s) {
7         label = s;
8         x = 1;
9     }
10    // ...
11 }
```

```
1 public class Bar extends Foo {
2     private int x;
3     // x in the subclass
4     public Bar() {
5         super("No name. :(");
6         x = 3;
7     }
8     // ...
9 }
```

```
1 Foo f = new Bar();
```

Now what is the value of x?

Duplicate fields in related classes (cont.)

In fact this creates two variables x for all Bar objects: one that's part of the Bar definition, and one that's inherited from the Foo class.

It's VERY BAD!



Like this is:

1

```
String Integer = new String("5"); // oh my.
```


The super constructor

When you create an object that is a subclass of another class (e.g., an Apple, which is a subclass of a Fruit), then you need to handle the constructor of the subclass specially.

You must construct the instance of the super class *first* and then do any other construction for the derived class.

To call the constructor for the super class you use the **super** keyword, such as here:

```
1 public class Fruit {
2     protected int deliciousness;
3     public Fruit(int d) {
4         deliciousness = d;
5     }
6 }
```

```
1 public class Apple extends Fruit
2     private float mass;
3         // in grams
4     public Apple() {
5         super(4);
6         size = 250;
7     }
8 }
```

To access something in the superclass, we use the reserved word **super**:

- `super()` calls the constructor of the superclass
- `super.myMethod()` calls the `myMethod()` method in the superclass.
- `super.myField` calls the `myField` field in the superclass (if one is defined there!)^[1]
- No, it's not possible to call `super.super()`.

^[1]Remember that the parentheses `()` denote a method.

If you don't write a constructor, Java provides one called the *default constructor*.

The default constructor takes no arguments and does nothing, and any instance variables that the class has are initialised to default values.

If you write any constructor then this default constructor will disappear, even if the new constructor you write takes arguments.