ECS510

Algorithms and Data Structures in an Object Oriented Framework

"ADSOOF"

Inheritance

The extended `DrinksMachine` examples

Class extension

`class Beta extends Alpha` means

- Objects of type `Beta` have all the variables and methods of objects of type `Alpha`
- They may also have additional variables and methods declared in class `Beta`
- An object of type `Beta` can be used anywhere where an object of type `Alpha` is required
- We refer to `Alpha` as the "superclass" of `Beta` and `Beta` as a "subclass" of `Alpha`

ExtDrinksMachine1

Example:

```
class ExtDrinksMachine1 extends DrinksMachine
{
 private ArrayList<Can> sprites;
```

- means an object of type `ExtDrinksMachine1` has all the variables of an object of type `DrinksMachine` plus an extra one, `sprites`
- It also has extra methods, representing a machine with a “Sprite” button as well as a “Coke” and “Fanta” button

`private` and `protected` variables

With `class Beta extends Alpha`

- Objects of type `Beta` have all the variables of objects of type `Alpha`, but methods in class `Beta` cannot access `private` variables from class `Alpha`
- They can access `protected` variables
- In `ExtDrinkMachine1` the extra method `pressSprite` needs to access the variables `price`, `balance` and `cash`, as well as `sprites`, so these would have to be declared as `protected` rather than `private`

`super` in constructors

- In a constructor for a subclass, the first statement `super(…);` means "use the code of the matching constructor of the superclass"
- Example

```
public ExtDrinksMachine1(int p, int c, int f, int s)
{
 super(p,c,f);
 sprites = new ArrayList<Can>();
 for(int i=0; i<s; i++)
    loadSprite(new Can("sprite"));
}
```

Actual and apparent type

With `class Beta extends Alpha` and

```
Alpha a;
Beta b;
```

- We can have `a=b;`
- We cannot have `b=a;`
- So a variable of type `Alpha` can refer to an object whose "actual" type is `Beta`
- `Alpha` is its "apparent" type

Subclass arguments

So with:

```
public static DrinksMachine
cheaper(DrinksMachine m1,DrinksMachine m2)
{
 if(m1.getPrice()<m2.getPrice())
      return m1;
 else
      return m2;
}
```

a call `cheaper(mach1,mach2)` could have either or both of `mach1` and `mach2` variables of type `ExtDrinksMachine1`

Casting

With `class Beta extends Alpha` and

```
Alpha a;
Beta b;
```

- We can have `b = (Beta) a;`
- This will throw an exception if `a` does not refer to an object of actual type `Beta`
- We can use `instanceof` to test for actual type:

```
if(a instanceof Beta) …
```

Calling extra methods

With `class Beta extends Alpha`

- A method which is in class `Beta` but not `Alpha` can only be called on a variable of type `Beta` (or a subclass of `Beta`)
- This applies even if a variable of type `Alpha` refers to an object of type `Beta`

Calling `pressSprite`

Example:

```
Can c;
if(m1 instanceof ExtDrinksMachine1)
   {
    ExtDrinksMachine1 em = (ExtDrinksMachine1) m1;
    c = em.pressSprite();
   }
else
   c = m1.pressFanta();
```

Overriding

- With `class Beta extends Alpha` we can also have full code for a method in `Beta` with the same signature as a method in `Alpha`
- This is referred to as “overriding” the method
- It can be used to create a subclass of objects which behave differently from normal objects of the superclass
- But objects of the subclass can still be used wherever objects of the superclass are expected

Dynamic Binding

- Suppose method `meth` from class `Alpha` is overridden in class `Beta`
- Now suppose

```
Alpha a;
Beta b;
…
a=b;
…
a.meth(…)
```

- Is the code used in this call the one in class `Alpha` or the one in class `Beta`?
- Answer is - the code from class `Beta`

`ExtDrinksMachine2`

- In this example, `ExtDrinksMachine2` overrides the `pressCoke` and `pressFanta` methods of `DrinksMachine`
- The idea is to represent a machine which contacts a supplier when it runs out of a particular drink
- When we press the “Coke” or “Fanta” button on a machine, we don’t know whether it is an ordinary one or one which contacts a supplier, this is like dynamic binding in code

Inheritance

- A way of re-using code - extend an existing class rather than write a completely new class
- A way of generalising - code for a superclass works for all subclasses
- Introduces some tricky concepts, in particular where a variable of a superclass refers to an object of a subclass

Concerns over Inheritance: the Fragile Base Class issue

- Inheritance was considered one of the most important aspects of object-oriented programming, but now most experts suggests it should be used cautiously
- The main issue of concern is that it creates a link between different pieces of code, the superclass and its subclasses, which can be hard to follow
- If the superclass is changed, this can have unexpected effects on its subclasses (the “fragile base class” issue).

Unexpected effects of overridden methods

- Another issue with inheritance comes from dynamic binding: if a method has a parameter of a particular class type, and is passed an argument of a subclass of that class, method calls it makes on the argument will use the code of the subclass if the subclass overrides the method
- All that it is necessary to override a method is to have the same header, but otherwise the code could perform a completely different operation
- This could cause problems if the code that took the subclass argument was written with expectations about how the methods of the argument would perform which are not met by the overriding code in the subclass
- It could even cause a breach in security, suppose the overriding code sent messages to report when methods are called to a spy

Overriding `equals`

- Dynamic binding is needed, for example when we call `obj1.equals(obj2)` we would want the code used to depend on the class of the object referred to by `obj1`
- However, we would expect the code used to perform in a way that fits in with the general idea of the `equals` method
- It should be the case that `obj2.equals(obj1)` should return the same as `obj1.equals(obj2)`
- It should be the case if `obj1.equals(obj2)` returns `true` and `obj2.equals(obj3)` returns `true` then `obj2.equals(obj3)` should return `true` as well
- There is nothing to stop us overriding `equals` with code that does not perform this way

The Liskov Substitution Principle

- The Liskov Substitution Principle (LSP) is named after Barbara Liskov, who first identified the problem
- The LSP says that methods should be overridden only by code that does not conflict with the specifications of the method it is overriding
- The LSP can be expressed in various ways, one is that an overriding method *"should not strengthen the preconditions or weaken the postconditions"*
- The preconditions of a method are any requirements on an argument, for example that an integer argument must not be negative or a list argument must not be empty
- The postconditions are the effects of a method call, what it returns or what mutating effects it has

Favour Composition over Inheritance

- Composition is when an object is made up of other objects
- A common piece of advice in programming is that you should "favour composition over inheritance"
- This means that rather than have one class as a subclass of another, you should consider whether it is better for it to have a field of the other class
- That means

```
class Beta { …
 private Alpha myAlpha;
```

rather than

```
class Beta extends Alpha
```

- Another way of putting this is that `Beta` "has-a" `Alpha` rather than `Beta` "is-a" `Alpha`

Delegation

- If you use composition rather than inheritance, a call of a method on a `Beta` object that requires action from the `Alpha` aspects would have to make a call on the `Alpha` object it refers to, this is called “delegation”
- If inheritance is used there is just a `Beta` object, no separate `Alpha` object, but with composition there is a separate `Alpha` and Beta object
- In some cases you might want there to be a single shared `Alpha` object for all `Beta` objects, or for several `Beta` objects to be able to share one `Alpha` object
- Aliasing like this is acceptable if it is an acknowledged aspect of the code design
- Aliasing an immutable object can be a useful way of saving on memory use (the “flyweight” design pattern)

Advantages of Inheritance

- Concerns over problems with inheritance do not mean it should never be used, but you need to be aware of the issues
- The problem of unexpected overriding can be removed by declaring a method as `final` (meaning it cannot be overridden) or a class as `final` (meaning no class can be declared as `extends` it, that is it cannot have subclasses)
- An advantage of having `class Beta extends Alpha` is that a `Beta` object can be passed to a method with an Alpha parameter, but that also applies with `class Beta implements Alpha`, where `Alpha` is an interface type (has only method headers)
- When `extends` is used, the shared code in the superclass is one way of meeting the "Don't repeat yourself" principle