Inheritance and Subtyping

1-20-2012
Opening Discussion

- Do you have any thoughts about what project you want to do?

- Minute essay comments:
  - IcPs and Assignments will focus on the projects with minor extensions.
  - Is making an RTS really feasible?
  - Will you code on the 2-D drawing outside of class?
  - Is it normal to feel overwhelmed and scared and want to change to a COMM major?
  - Project analysis and design is mostly you, but you can talk to me.
How complex can use-case diagrams get?
Is it easier to do fun projects or useful projects?
Pseudo-networking through shared files.
I lot of this semester focuses on the idea of abstraction.

We saw a little abstraction in the first semester, like having the ability to pass in different functions to perform different operations.

This capability can be expanded greatly and gives us a lot of power.

Remember that we don't want to write more code than we have to. In particular, duplicating code is very risky.
Polymorphism

- Literally means “many shapes”. In programming means “many types”.
- Our code to date has been monomorphic. It worked with only one type.
- We can add a lot of flexibility with polymorphism.
  - Write something once and have it work with many types.
- Universal polymorphism → works with infinite number of types
One from of Universal Polymorphism is called Inclusion Polymorphism.

We get this when we have that ability to say that one type is a subtype of another type.

If B is a subtype of A, then any code that wants an instance of A can use an instance of B.

Consider the type Fruit.
The way we get inclusion polymorphism in Scala is through inheritance.

- class B(...) extends A(...) { ... }

This means B gets all the stuff from A.

- For that reason, it can be safely used as a subtype.
- Subtypes can override implementations.

Only use inheritance to represent an “is a” relationship. Even then don't use it unless it makes sense.

Let's consider the example of a shape.
Details

- **Visibility**
  - Private – subtypes can't get access, but have a copy of data.
  - Protected – subtypes can access, but other things can't.

- Call methods on supertypes with `super.method(…)`

- You can only inherit from one class.

- In UML this is shown as an open headed arrow from the subtype to the supertype.
It turns out you have been using inheritance for a long time.

- val frame = new MainFrame { … }

This code actually makes a new class with no name. It is a subtype of MainFrame.

Because it doesn't have a name of its own it is called an anonymous class.
Abstract Classes

- Often a supertype needs to have a method, but there is no general implementation.
- In this case, the method should be abstract. That simply means it isn't implemented.
- Classes with abstract members need to be declared abstract.
  - abstract class Shape {
    - def area:Double
  }
- Abstract classes can't be instantiated.
Traits

- A trait is like an abstract class that can't take arguments.
- You can inherit from multiple traits.
  - class B extends A with T1 with T2 … { … }
- If methods are duplicated, it searches for the one to use starting at the end of the list and working backward (plus some other details).
Sometimes you have methods that shouldn't be overridden or classes that shouldn't be inherited from. In that case you make them final.

For example, immutable classes need to be final so that people can't make mutable subtypes.
As you know, functions are used in many places in Scala.

If you provide an apply method, you can have your class inherit from a function type.

This would let you pass instances of your function into methods that want functions.
Let's write some code for the drawing program.
Questions?

How might inheritance and subtyping be used in your project?

Next class we will use this in our project.