cs2220: Engineering Software

Class 11:
Subtyping and Inheritance

Fall 2010
University of Virginia
David Evans

Schedule Updates

PS4 is now due on **Monday, October 11**
(October 12: Reading day)

Start thinking about project ideas

Once you have an idea for your project, can substitute parts of your project for programming parts of PS

Kinds of Abstraction

Procedural Abstraction

Abstraction hides details of computations

One procedure abstracts many information processes

Abstraction by Specification

Abstraction hides how a computation is done

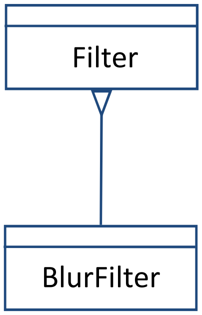
One specification can be satisfied by many procedures

Data Abstraction

Abstraction hides how data is represented

One datatype can be implemented many ways

Subtyping



BlurFilter is a ***subtype*** of Filter
Filter is the ***supertype*** of BlurFilter
BlurFilter $\subseteq$ Filter

Subtype Abstraction allows us to abstract **many possible datatypes** with their supertype.

Subtype Substitution

If *B* is a subtype of *A*, everywhere the code expects an *A*, a *B* can be used instead.

```
Filter f = new BlurFilter();
```

bf= (Blur Filter) f;

```
Filter f;
BlurFilter bf;
...
f = bf;
```

~~bf = f;~~

Applying a Filter

```
// EFFECTS: Returns a Filter object
//   associated  with the input name.
private Filter loadFilter(String fname);
```

```
Filter f = loadFilter(command);
int idx = images.getSelectedIndex();
if (idx < 0) {
  reportError("An image must be selected to apply an effect.");
  return;
}
f.setImage(workingImages.get(idx), (String) imagesModel.get(idx));
Image result = f.apply();
if (result == null) {
  reportError("Error applying filter");
} else {
   addImage(result, f.getImageName() + "/" + f.getFilterName());
}
```

from ps4/GUI.java

Supertype Specification

```
public abstract class Filter {
  // OVERVIEW: A Filter represents an image and provides a technique for altering it.
  //    A Filter may be in one of three states: uninitialized, initialized,
  //    and applied.  An initialized or applied filter has an associated image;
  //    and a Pixels object that represents the pixel data (possibly modified
  //    by the filter) in the image.
 public Filter()
     // EFFECTS: Initializes this to an unintiali
 final public void setImage(Image p_image,
    // REQUIRES: this is uninitialized
    // MODIFIES: this
   // EFFECTS: Sets the image for this to p_image; sets this to the initialized state.
 public String getImageName()
    // EFFECTS: Returns the image name associated with the filter.
 public String getFilterName()
    // EFFECTS: Returns the name of the filter.
  …
```

All subtypes must implement the supertype's specification. But, they can provide different implementations.

Method Dispatch

Assume B is a *subtype* of A

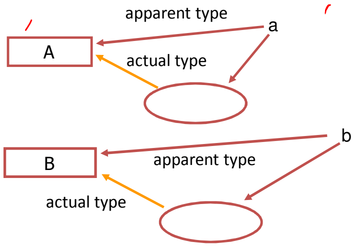
If both A and B have a method filter which method should be called?

```
A a = new A ();
B b = new B ();

a.filter();          Calls class A's filter method
b.filter();          Calls class B's filter method
a = b;
a.filter()           Calls class B's filter method
```

Dynamic Dispatch

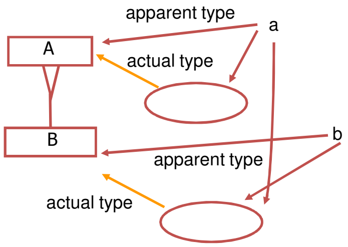
Search for the method up the type hierarchy, starting from the **actual** (dynamic) **type** of the object



```
A a = new A ();
B b = new B ();

a.filter();
b.filter();
```

Dynamic Dispatch



```
A a = new A ();
B b = new B ();

a.display ();
b.display ();
a = b;
```

Now: **apparent type** of a is A,
actual type of a is B

Apparent and Actual Types

Apparent types are associated with declarations

Never change

Actual types are associated with objects

Always a subtype of the apparent type

Can change which subtype it is

Compiler does type checking using apparent type

JVM does method dispatch using actual type

How can we change the **actual type** of a variable?

How can we change the **apparent type** of an expression?

Downcasting

```
Filter f = new Filter();
BlurFilter bf = new BlurFilter();

f = bf;

bf = f;                      Compiler type mismatch error

bf = (BlurFilter) f;

bf = (AddFilter) f;          ClassCastException
```

Casting changes the *apparent* type. The VM must check at runtime that the actual type is a subtype of the cast type (if not, ClassCastException).

Implementing a Subtype

```
public abstract class Filter {
  ...
  public String getFilterName() {
    return "basic";
  }
  ...
}
```

Supertype

```
public class BlurFilter extends Filter {
  ...
  @Override
  public String getFilterName() {
    return "blur";
  }
  ...
}
```

Subtype

Dynamic Dispatch

```
Filter f = loadFilter(command);
int idx = images.getSelectedIndex();
if (idx < 0) {
  reportError("An image must be selected to apply an effect.");
  return;
}
f.setImage(workingImages.get(idx), (String) imagesModel.get(idx));
Image result = f.apply();
if (result == null) {
  reportError("Error applying filter");
} else {
   addImage(result, f.getImageName() + "/" + f.getFilterName());
}
```

from ps4/GUI.java

Overriding Methods

```
public abstract class Filter {
  ...
  protected abstract void filter();
    // REQUIRES: this must be initialized
    // MODIFIES: this
   // EFFECTS: alters the image in a manner specified by the filter.
  ...
}
```

```
public class BlurFilter extends Filter {
   ...
  @
  pr
    /
   //
   //
  {

   }
}
```

```
public class FlipFilter extends Filter {
   @Override
```

```
public abstract class MultiFilter extends Filter
{
   .
   p
}
```

```
public class AddFilter extends MultiFilter {
   ...
@Override
protected void filter()
   // MODIFIES: this
   // EFFECTS: Replaces each pixel in the image with the
   //   bitwise or of the corresponding pixels in all the images.
```

Subtyping vs. Inheritance

Inheritance

Reusing the implementation of one type to build a new datatype

Subtyping

Defining a new type that can be used everywhere the supertype is expected

These are very different notions, but often confused! It is possible to have inheritance without subtyping, and to have subtyping without inheritance.

Subtyping/Inheritance in Java

extends: both subtyping and inheritance

implements: just subtyping

class B **extends** A { ... }
  B is a subtype of A
  B inherits from A
class C **implements** D { ... }
  C is a subtype of D

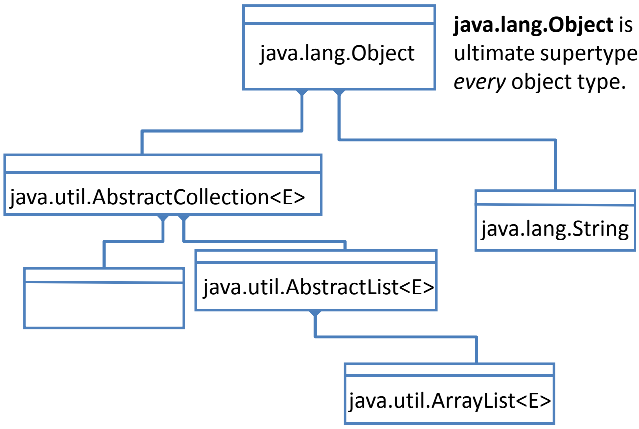
Is it possible to get inheritance without subtyping?

```
public class A {
   // rep is a B
   private B rep;

   public A() { rep(); }
   public int method(int x) { return rep.method(x); }
   ... // same for all B methods you want to “inherit”
}
```

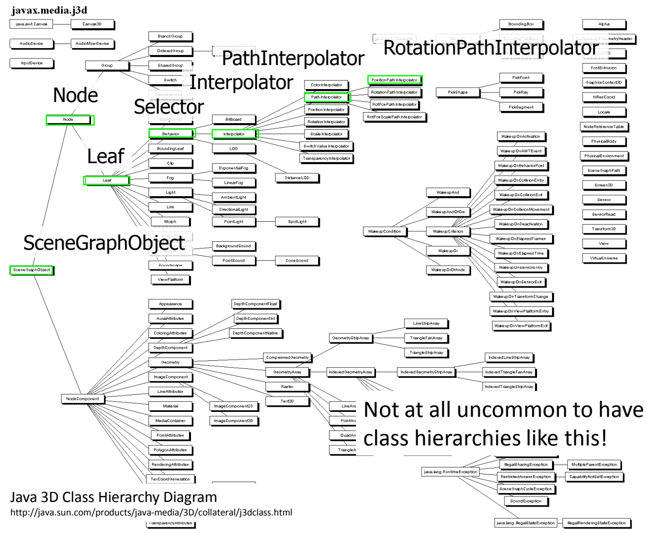
Not conveniently. But, this reuses most of B’s implementation without allowing A objects to be used where B is expected.

Java's Type Hierarchy



java.lang.Object is the ultimate supertype of *every* object type.

http://download.oracle.com/javase/6/docs/api/java/util/TreeSet.html



Not at all uncommon to have class hierarchies like this!

Java 3D Class Hierarchy Diagram
http://java.sun.com/products/java-media/3D/collateral/j3dclass.html

All Classes are Subtypes

```
public class Graph {
  …
}
```

really means:

```
public class Graph extends Object {
  …
}
```

```
public class java.lang.Object {
  public boolean equals(Object o) { ... }
  public String toString() { ... }
  ... // 7 other methods
}
```

Why Subtyping is **Scary**

Reasoning about correct code now requires thinking about all possible subtypes!

Substitution Principle (Behavioral Subtyping): imposing limits on the possible specifications of subtypes to make this possible!

Charge

Subtyping

- Allow one type to be used where another type is expected

Inheritance

- Reuse implementation of the supertype to implement a subtype

Thursday:

- When is it safe to say B is a subtype of A?

Now: project ideas!