CSSE 220

Interfaces and Polymorphism

Check out *Interfaces* from SVN

Object-Oriented Programming

- The **three pillars** of **Object-Oriented Programming**
  - Encapsulation (already covered)
  - Inheritance (we will partly cover it today)
  - Polymorphism (also covered today)

Interfaces – What, When, Why, How?

- What:
  - Code Structure used to express operations that multiple class have in common
  - No method implementations
  - No fields
- When:
  - When abstracting an idea that has multiple, different implementations

Open simpleExample

Interfaces – What, When, Why, How?

- Why:
  - Provide method signatures and documentation
  - Create a **contract** that someone must follow
    - Client Code Reuse, for example, Java Event Handlers
- How:

```
public interface InterfaceName {
    //method definitions
    //We'll look more closely at the syntax in a later slide
}
```

Interface Types: Key Idea

- Interface types are like **contracts**
- A class can promise to **implement** an interface
  - MUST implement every method
  - Client code knows that the class will have those methods
    - Compiler verifies this
  - Any client code designed to use the interface type can automatically use the class!
- Interfaces help to **reduce coupling** by tying your design to the interface and not the class implementation.
  - A new interface implementation can be switched out for the original without changing the rest of the code

Q1

Interface Types can be used anywhere that a class type is used.

- Once an interface is defined, it can be used as a type.
- Say we have an interface named Pet, and Dog and Cat implement this interface…

1. Variable Declaration:
   - Pet d = new Dog();
   - Pet c = new Cat();
2. Parameters:
   - public static void feedPet(Pet p) {…}
   - Can call with any object of type Pet:
     - feedPet(new Dog());
     - feedPet(new Cat());

Interface Types can be used anywhere that a class type is used.

(…continued from last slide)

3. Fields:
   - private Pet pet;
4. Generic Type Parameters:
   - ArrayList<Pet> pets = new ArrayList<Pet>();
   - pets.add(new Dog());
   - pets.add(new Cat());

NumberSequence Example

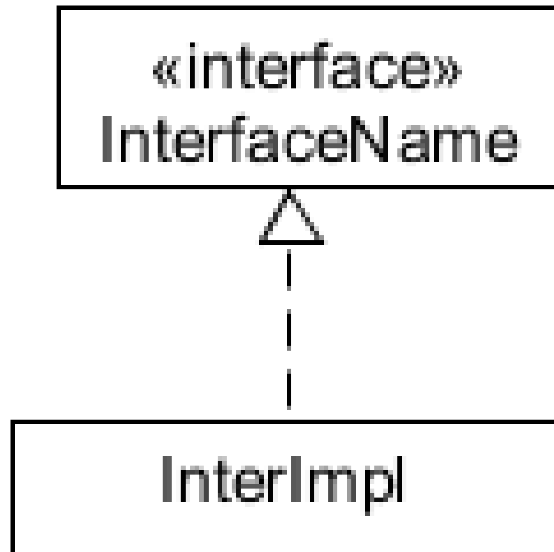
Why is this OK?

```
Pet p = new Dog();
p.feed();
p = new Cat();
p.feed();
p = new Pet(); // NO!
```

- Any child type may be stored into a variable of a parent type, but not the other way around.
  - A Dog is a Pet, and a Cat is a Pet, but a Pet is not **required** to be a Dog or a Cat.
  - And how could you ***construct*** a Pet?
- But how does Java know which method implementation to use?

Q2

Notation: In UML

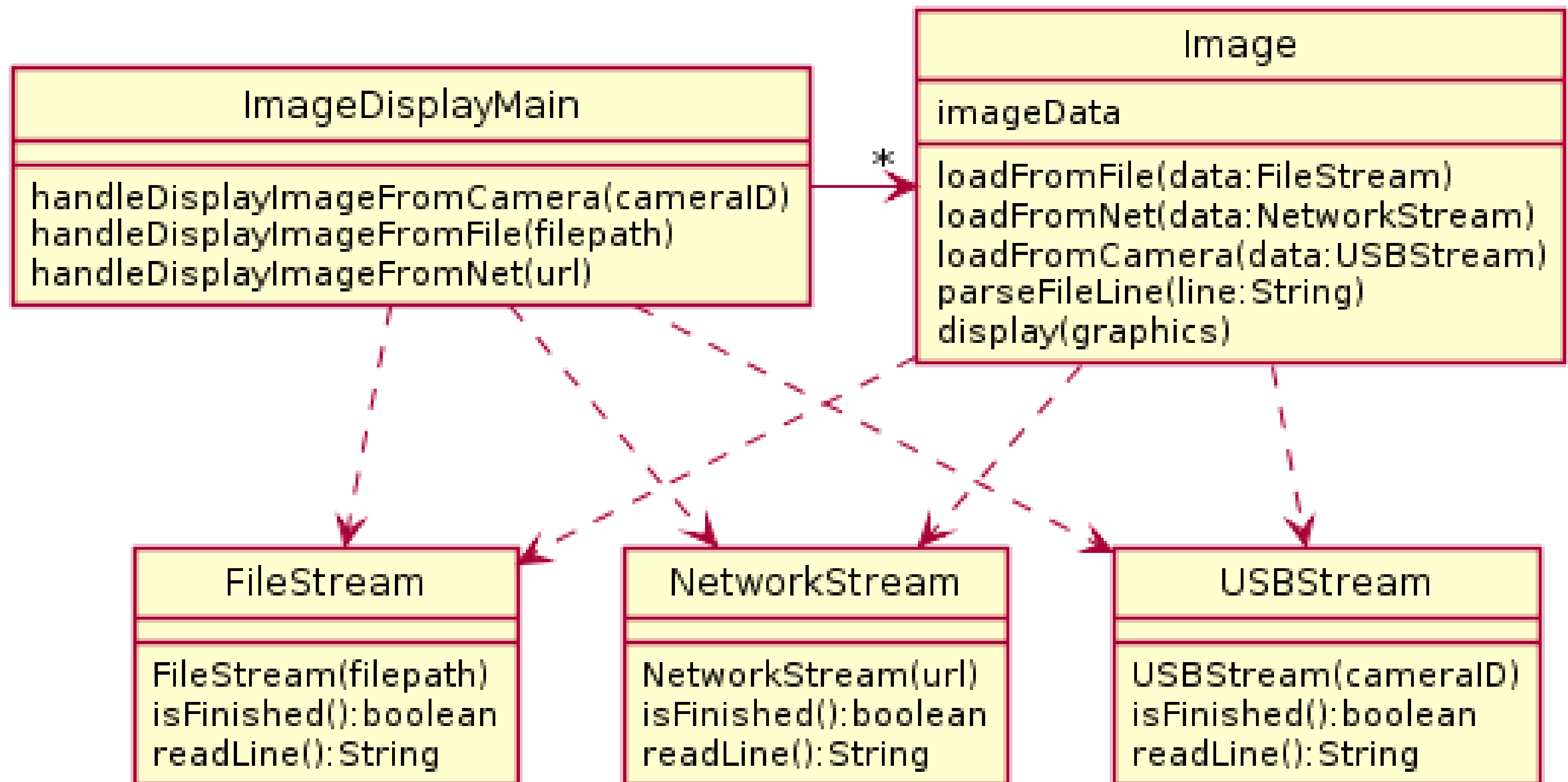


- Closed triangle with a dashed line in UML is an “is-a” relationship
- Read this as:

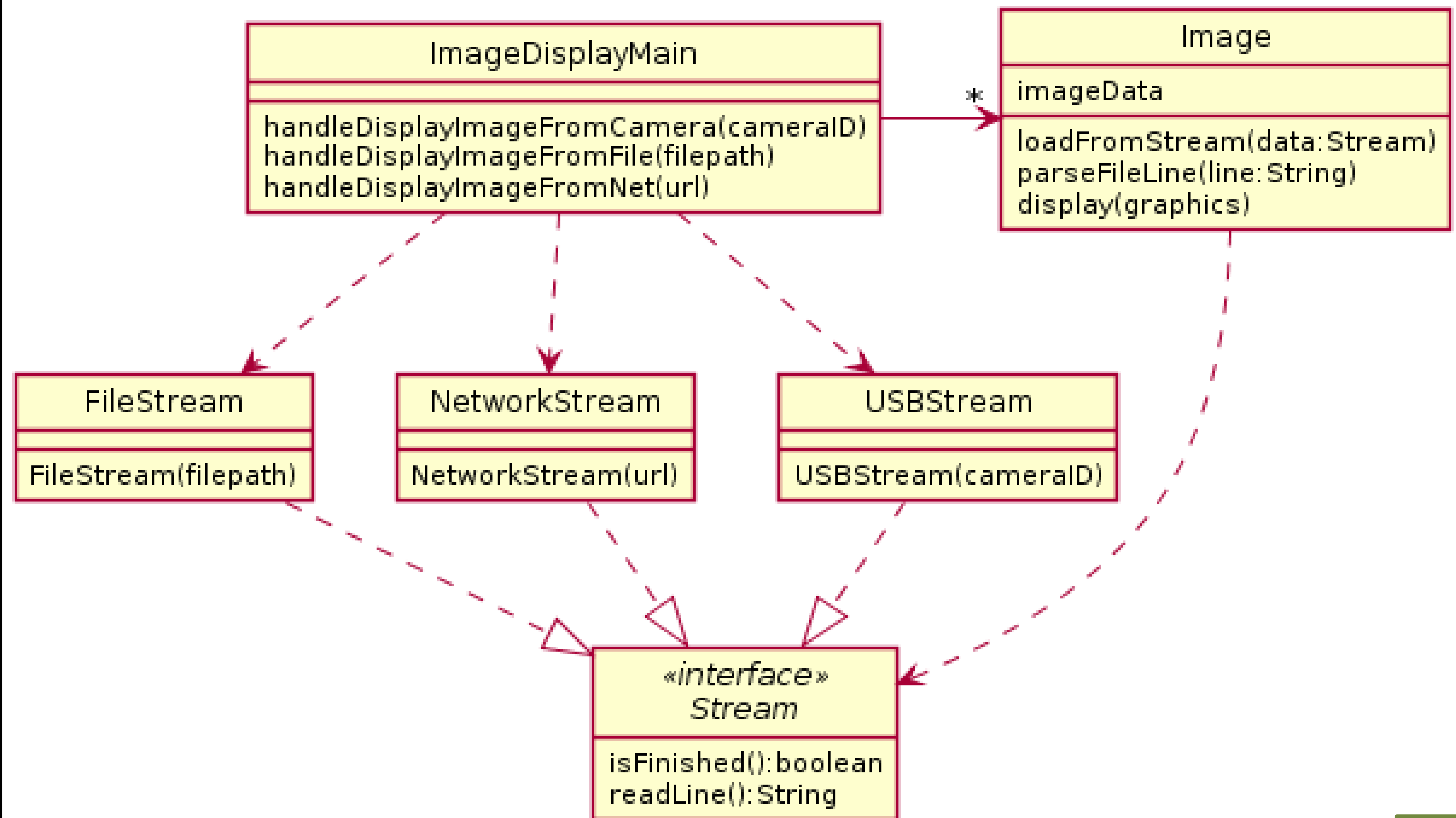
  InterImpl is-an InterfaceName

Q3

A particular Image class loads image files from a variety of sources and display them on the screen. The image sources can be a file, a network link, or a USB camera. All the various sources have similar functions, but because they are different types they each require a specialized function. Show how an improved approach using interfaces can remove the code duplication.



Solution



Q4

Polymorphism! (A quick intro)

- Origin:
  - Poly → many
  - Morphism → shape
- Classes implementing an interface give **many differently “shaped” objects for the interface type**

- Java knows what method implementation to use thanks to:
  - Late Binding:
    - choosing the right method based on the actual type of the implicit parameter (variable before the dot) **at run time**
  - For the p.feed() example:
    - Java decides at runtime which implementation to use based on the type of the object instance.
    - The Dog’s feed method may specify dog food, and the Cat’s may specify cat food.

Q5

Notation: In Code

```
public interface InterfaceName{
    /**
     *  regular javadocs
     */
    void methodName(int x, int y);

    /**
     *  regular javadocs here
     */
    int doSomething(Graphics2D g);
}

public class InterImpl implements InterfaceName {

}
```

interface, not class

Automatically public, so we don't specify it

No method body, just a semi-colon

InterImpl promises to implement all the methods declared in the **InterfaceName** interface

Refactoring to an Interface

- stringTransforms package
  - Review the code in the stringTransforms package
  - Attempt to refactor the given code using an interface by thinking about what operation is performed repeatedly
  - There is a hint at the bottom if you're not quite sure where to start, but only use it if you need

How does all this help reuse?

- Can pass an **instance** of a class where an interface type is expected
  - But only *if the class* ***`implements`*** *the interface*

- We could add new functions to a NumberSequence's abilities without changing the runner itself.
  - Sort of like application "plug-ins"

- We can use a new TransformInterface without changing the method that uses the TransformInterface instance

- **Use interface types** for field, method parameter, and return types whenever possible. Like Pet instead of Dog, and List for ArrayList.
  - **List**<Pet> pets= new ArrayList<Pet>();