Polymorphism

- “Have many forms”
- A polymorphic reference refers to different objects (related through inheritance) at different times

Suppose class Plane extends Vehicle

Vehicle mover; //a Vehicle reference
Plane flyer;   //a Plane reference
mover= new Vehicle(...);
flyer= new Plane(...);
// A plane is a vehicle
  mover= new Plane(...);
mover= flyer;
// A vehicle is not a plane
  flyer= new Vehicle(...);  //invalid

Another polymorphic example

Vehicle[] mover = new Vehicle[5];
mover[0] = new Vehicle(...);
mover[1] = new Plane(...);
mover[2] = new Plane(...);
mover[3] = mover[1];

Accessing methods/variables through polymorphic references

Ask two questions:
1. What determines whether a method/variable can be accessed?
2. For an overridden method, what determines which version gets invoked?

Accessing methods/variables through polymorphic references

The type of the reference determines the methods and fields that can be accessed

class V {
    int num1;
    void vmethod() { num1++; }
}
class W extends V {
    int num2;
    void wmethod() { num2++; }
    //wmethod cannot be called thru reference
    //of type V
}
Client code:

```java
V x = new W();
System.out.println(x.num1); //?
System.out.println(x.num2); //?
x.vmethod(); //?
x.wmethod(); //?
```

Accessing *overridden* methods through polymorphic references

- The *type of the object* determines which version of the method gets invoked
- Class `Vehicle` has method `toString` that class `Plane` overrides:

```java
Vehicle v1 = new Vehicle(...);
Vehicle v2 = new Plane(...);
System.out.println(v1.toString()); //Vehicle’s version
System.out.println(v2.toString()); //Plane’s version
```

Recap—extending a class

- Subclass is a more specific version of the superclass
- Subclass inherits public and protected members from superclass
- You can declare new variables and methods in subclass
- You can override inherited methods with new definition
- Do not re-declare variables in subclass
  Called *shadowing*—not good practice in general

```java
class Room {
    private static int nextID = 1;
    // id of next room to be created
    protected int id;
    protected int mess; // mess level
    public Room(int mess) { this.mess = mess;
        id = nextID; nextID++; }
    public String toString() { return "Room " + id; }
    public void clean() { mess--;
        if (mess<0) mess=0; }
    public void report() { System.out.println(toString() +
        ", has mess level "+ mess); }
    public static void countRooms() {
        System.out.println((nextID-1) + " rooms in total"; }
} //class Room
```

class Bathroom extends Room {
    private boolean hasShower;
    public Bathroom(int mess, boolean hasShower) {
        super(mess);
        this.hasShower = hasShower;
    }
    public String toString() {
        String line = super.toString();
        line += ", a bathroom";
        if (hasShower) line += " with shower";
        return line;
    }
    public void majorCleanUp() {
        clean(); clean(); clean(); clean();
    }
} //class Bathroom

*static* methods & variables

- Do not re-declare *static* components!
- Same rules for inheritance (accessibility) with respect to visibility modifiers
  - Method: implicitly *final*
  - Variable: same memory space as superclass
Client code:

```
Room r1 = new Room(5);
Bathroom r2 = new Bathroom(10, true);

// Method invocation
System.out.println(r1);
System.out.println(r2);
r1.report();
r2.report();
System.out.println();
r1.clean(); r1.report();
r2.clean(); r2.report();
r2.majorCleanUp(); r2.report();
//r1.majorCleanUp();
System.out.println();
```

More client code:

```
// Polymorphism
Room r3 = new Bathroom(20, false);
System.out.println(r3); // ??
r3.clean(); // ??
r3.report(); // ??
r3.majorCleanUp(); // ??
r3.report();

// Static methods and variables
Room.countRooms();
Bathroom.countRooms();
```

Important ideas in inheritance

- Use different hierarchies for different problems
- Single inheritance
- Keep common features as high in the hierarchy as reasonably possible
- Use the superclass’ features as much as possible
- Inherited features are continually passed down the line
- “Inherited” ⇒ “can be accessed as though declared locally”
  (private variables in superclass exist in subclasses; they just cannot be accessed directly)

The Object class

If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the Object class
⇒ All classes are derived from the Object class

```
class Room
is the same as

class Room extends Object
```

abstract class

- A placeholder in a class hierarchy that represents a generic concept
- Cannot be instantiated
- Modifier: abstract
  public abstract class Geometry
- Can contain abstract methods
  public abstract double Area();
- Subclasses of abstract classes will “fill out” these abstract methods