Recall that we identified three key elements of OO programming:

1. Encapsulation
2. Subtyping
3. Inheritance

We are now ready to discuss the third of these, inheritance. Inheritance is a mechanism that supports extensibility and reuse of code, both across programs and within programs.

1 Motivating example

Suppose we get an implementation of the Sam Loyd Puzzle from a code supplier. In this implementation, the puzzle is represented by a 2D array of integers:

```java
class APuzzle {
    int[][] tiles;
    public int tile(int r, int c) {
        return tiles[r][c];
    }
    public void move(Direction d) { ... }
}
```

Now suppose that we want to build a better puzzle implementation by reusing this code. For example, suppose that we want the puzzle to keep track of how many moves have been made. We might proceed by copying the code of `APuzzle` to create a new class `LogPuzzle` which we then add new fields and methods to. But this strategy doesn’t work very well when the original supplier fixes some bugs in `APuzzle` and gives us `APuzzle'`. We’ll either have to apply the bug fixes to `LogPuzzle` or our upgrades to `APuzzle'`, as illustrated in Figure 1. Either one will be hard to do in an automatic way.

![Figure 1: The problem of merging upgrades](image)

At least for some kinds of changes that we might make between `APuzzle` and `LogPuzzle`, inheritance offers a solution to this problem. We can think of inheritance as a language mechanism for copying a class and making certain kinds of changes to that class.

To add the new functionality to `LogPuzzle`, we inherit from `APuzzle`:
class LogPuzzle extends APuzzle {
    private int num_moves;
    public int numMoves() return num_moves;
    public void move(Direction d) {
        num_moves++;
        super.move(d);
    }
}

Here APuzzle is the superclass and LogPuzzle is the subclass. In general, classes inheriting from each other form an inheritance hierarchy or class hierarchy, in which superclasses are above their subclasses.

The LogPuzzle class is just like the APuzzle class, except:

• It has an additional method, numMoves, and a a new field, num_moves.
• It has a new version of the move that overrides the original version from APuzzle. It overrides the original because it has the same name and signature as the original method. The signature consists of the types of all of the arguments and the return type.
• The type LogPuzzle is a subtype of the type APuzzle. That is, LogPuzzle <: APuzzle.

2 Method dispatch and inheritance

Because of the subtyping relationship between the two classes, we can write the following code:

APuzzle p = new LogPuzzle();
p.move(up); // which move?

Which version of the move is used? The static type of p is APuzzle, but the dynamic type is the object’s class, LogPuzzle. For ordinary, non-static methods, the dynamic type defines what method is used. So it is the LogPuzzle version that is run. This is known as run-time method dispatch.

There are two different ways to explain how inheritance works when a method is invoked. One way, which you have probably already heard in the past, is to say that when a method is invoked, the system searches upward through the inheritance hierarchy looking for the first implementation of the method. But it is equivalent to say that the methods from the superclasses are copied down to their subclasses except when overridden.

To better understand how inheritance operates, suppose that the superclass APuzzle had a method scramble that used move:

class APuzzle {
    void scramble() {
        ... move(random_dir) ...
    }
}
APuzzle p = new LogPuzzle();
p.scramble();
// is p.numMoves() equal to 0?

When the method scramble() calls move(), which version is called? We can answer this question by understanding that the code of scramble acts as though it is copied to the run-time class of p, which is LogPuzzle. Therefore, when the method move() is called, the LogPuzzle version of move() is used.
3 Inheritance and static methods

Static methods complicate the story slightly, because they cannot be overridden by subclasses. The choice of what method to call is made in the class in which the method call is made. It does not change when the code making the call is inherited into a subclass. Consider the following code:

```java
class A {
    static int f() { ... }
    void g()
        f();
}
class B extends A {
    static int f() { ... }
}
A x = new B();
x.g();
```

When the method `g()` is called, the call to `f()` goes to the `A` version of `f` rather than the `B` version. Because `f` is a static method, the call to `f()` is exactly the same as if it were written `A.f()`. Therefore it does not change in meaning when it is inherited by `B`.

The special syntax `super.move()` , which was used in the implementation of `LogPuzzle` above, is also a static call that always goes to the `APuzzle` version of `move`.

4 Constructors

Constructors are special static methods that are used to ensure that objects are fully initialized before they are used. If a class defines one or more constructors, new objects of that class can only be created using the constructor. Therefore the constructor for any subclass must call a superclass constructor, as in the following example:

```java
class APuzzle {
    public APuzzle(int size) {
        tiles = new int[size][size];
    }
}
class LogPuzzle extends APuzzle {
    public LogPuzzle() {
        super(4);
        num_moves = 0;
    }
}
```

Here, the constructor `LogPuzzle` always creates puzzles of size 4×4, which is accomplished by calling the superclass constructor with `super(4)`. The call to the superclass constructor is static.

5 Protected visibility

What if we want the `LogPuzzle` code to access the `tiles` field directly? As defined we cannot, because `tiles` is declared as private. However, if we give `tiles` the protected visibility in `APuzzle`, it becomes visible to all subclasses of `APuzzle`:
Protected fields and methods form a second interface to a class. Public methods and fields are the public interface, which is exposed to client code. Protected methods and fields are the specialization interface, which is available to subclasses but not to ordinary clients. One of the challenges of good object-oriented design is to design both of these interfaces effectively, without confusing their roles. Designing a good specialization interface is especially important for object-oriented libraries where the classes provided by the library are intended to be extended through inheritance.

6 Protected methods and the specialization interface

Suppose that scramble() had been defined to call a protected method internal_move instead of the public move method:

```java
class APuzzle {
    public void scramble() {
        ... internal_move(n); ...
    }
    protected internal_move(int d) { ... }
}
class LogPuzzle extends APuzzle {
    protected internal_move(int d) {
        num_moves++;
        super.internal_move(d);
    }
}
```

This example shows that the specialization interface of APuzzle allows the LogPuzzle class to change the behavior of existing public methods without overriding them directly. Protected methods are hooks for future extensibility of OO code. The specialization interface defines how code can be extended.

7 Visibility and overriding

Protected and public methods can be overridden by subclasses, whereas private methods cannot. When a method is overridden, its visibility can be changed to make the method more visible than in the superclass.
Changing in the opposite direction would not make sense, because if the subclass restricted visibility, it
could always be casted up to the superclass type where the original visibility would hold. In any case, the
visibility of a method ordinarily remains the same when it is overridden.

8 Acknowledgments

These notes build on slides by Dexter Kozen and others.