Object Oriented Software Design

Polymorphism, Abstract Classes, Interfaces

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

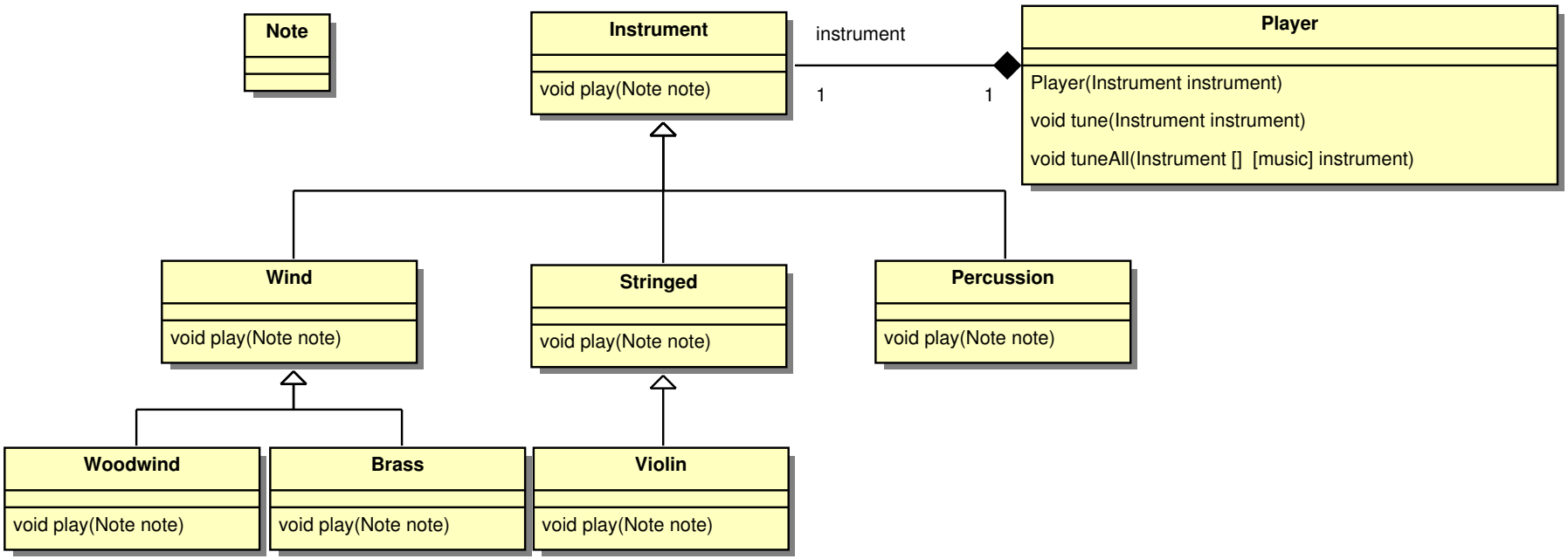
October 28, 2010

Outline

Outline

Using Inheritance

- Let's complete the example with the instruments



The Note class

- Let's start from the Note class

music/Note.java

```
package music;

public class Note {
    private String noteName;
    private Note(String noteName) {
        this.noteName = noteName;
    }
    public String toString() { return noteName; }
    public static final Note
        C = new Note("C"),
        C_SHARP  = new Note("C Sharp"),
        D   = new Note("D"),
        D_SHARP   = new Note("D Sharp"),
        E = new Note("E"),
        F = new Note("F");
    // Etc.
} ///:~
```

The Note class

- The constructor is **private**
  - This means that the user of the class cannot create any object of class Note
- Also, there is only method **toString()** in the public interface
  - The only thing that we can do is to use the public static members (`C`, `D`, `E`, etc.) that are all **final** (i.e. they are constant), and convert them to String.
- This the the standard way to create a set of constants with Java
  - Similar to the enum in C/C++

Class Instrument

- Instrument is the base class of our hierarchy

music/Instrument.java

```
package music;

public class Instrument {
    public void play(Note n) {
        System.out.println("Instrument.play() " + n);
    }
    public String what() { return "Instrument"; }
    public void adjust() {}
}
```

The Player class

music/Player.java

```
package music;

public class Player {
    String name;

    public Player(String name) {
        this.name = name;
    }

    public String toString() {
        return "Player : " + name;
    }

    public void tune(Instrument i) {
        i.play(Note.C);
    }

    public void tuneAll(Instrument[] e) {
        for(int i = 0; i < e.length; i++)
            tune(e[i]);
    }
}
```

The main

MusicPlayer.java

```
import com.bruceeckel.simpletest.*;
import music.*;

public class MusicPlayer {
    public static void main(String[] args) {
        Player player = new Player("Myself");
        Instrument[] orchestra = {
            new Wind(),
            new Percussion(),
            new Stringed(),
            new Brass(),
            new Woodwind()
        };
        player.tuneAll(orchestra);
```

What happens?

- The **Player** does not know about the existence of any of the Instrument classes
  - it calls the **play()** method of **Instrument**
  - and the **play()** method of the corresponding object is called
  - When the `Player.java` is compiled, how can the compiler know which function to call?
- This mechanism is called **dynamic binding**
  - At the time of compiling `Player.java`, the call is substituted by a simple code that looks into a table to understand with method to call
  - We will see how the **virtual table** mechanism works when we will study C++
  - Right now, just remember that in Java you only have dynamic binding (and this causes some extra overhead in function calls)

Polymorphism

- This mechanism is also called **polymorphism**, and Java methods are said to be **polymorphic**
- Now, let's try to change the list of instruments
- Also, let's try to add a new **Instrument** (e.g. a **Guitar**)
- Which code needs to be changed?
  - Not the **Player** class
  - Not the **Instrument** class
  - Only the **MusicPlayer** class which creates the **Guitar** object!
- We added a new behaviour with minimal changes
  - The changes can even be less than that!
- We are not always so lucky
  - Sometimes is not easy to minimise changes
  - For example, in some cases we **must** know the object type!

Outline

Classes you cannot instantiate

- Let’s continue with the **Instrument** example
  - Does it make sense to create an object of a type **Instrument**?
  - Does it make sense to call the **play()** method of the base class **Instrument**?

Classes you cannot instantiate

- Let's continue with the **Instrument** example
  - Does it make sense to create an object of a type **Instrument**?
  - Does it make sense to call the **play()** method of the base class **Instrument**?
  - No, actually there is not such a thing as an **Instrument**
  - There are many *types of Instrument*
- This is where Abstract classes are useful
  - We want to tell the language that it is not possible to create an instrument, and call its method
  - Instrument represent an interface
- You do so by saying that a method is **abstract**, i.e. it has not implementation
  - You also have to say that the class is abstract

Abstract Instrument

- Let's see the abstract version of our Instrument class:

music2/Instrument.java

```
package music2;

public abstract class Instrument {
    abstract public void play(Note n);
    public String what() { return "Instrument"; }
    public void adjust() {}
}
```

Abstract Wind

Of course, also Wind, Percussion and Stringed must be abstract:

music2/Wind.java

```
package music2;

public abstract class Wind extends Instrument {
    abstract public void play(Note n);
    public String what() { return "Wind"; }
    public void adjust() {}
}
```

music2/Stringed.java

```
package music2;

public abstract class Stringed extends Instrument {
    abstract public void play(Note n);
    public String what() { return "Stringed"; }
    public void adjust() {}
}
```

The main class

- Notice that this time we cannot create Wind and Percussion instruments:

MusicPlayer2.java

```
import com.bruceeckel.simpletest.*;
import music2.*;

public class MusicPlayer2 {
    public static void main(String[] args) {
        Player player = new Player("Myself");
        Instrument[] orchestra = {
            new Woodwind(),
            new Violin(),
            new Brass(),
            new Woodwind()
        };
        player.tuneAll(orchestra);
```

Outline

Completely abstract classes

- **Instrument** is now an abstract class
  - It contains normal methods (with code), and one abstract method without code
- Sometimes it is useful to have only interfaces, i.e. classes where you do not provide any code at all
  - In Java this is done with the **interface** keyword

Instrument interface

- Here is how Instrument is transformed into an interface:

music3/Instrument.java

```
package music3;

public interface Instrument {
    int I = 5; // this is static and final
    // all methods are public by default
    void play(Note n);
    String what();
    void adjust();
}
```

How to use Instrument

- With interfaces, you have to use the keyword **implements** instead of **extends**

music3/Wind.java

```
package music3;

public abstract class Wind implements Instrument {
    abstract public void play(Note n);
    public String what() { return "Wind"; }
    public void adjust() {}
}
```

music3/Stringed.java

```
package music3;

public abstract class Stringed implements Instrument {
    abstract public void play(Note n);
    public String what() { return "Stringed"; }
    public void adjust() {}
}
```

The main class

- Notice that you can declare a reference to an interface:

MusicPlayer3.java

```
import com.bruceeckel.simpletest.*;
import music3.*;

public class MusicPlayer3 {
    public static void main(String[] args) {
        Player player = new Player("Myself");
        Instrument[] orchestra = {
            new Woodwind(),
            new Violin(),
            new Brass(),
            new Woodwind()
        };
        player.tuneAll(orchestra);
```

Multiple interfaces

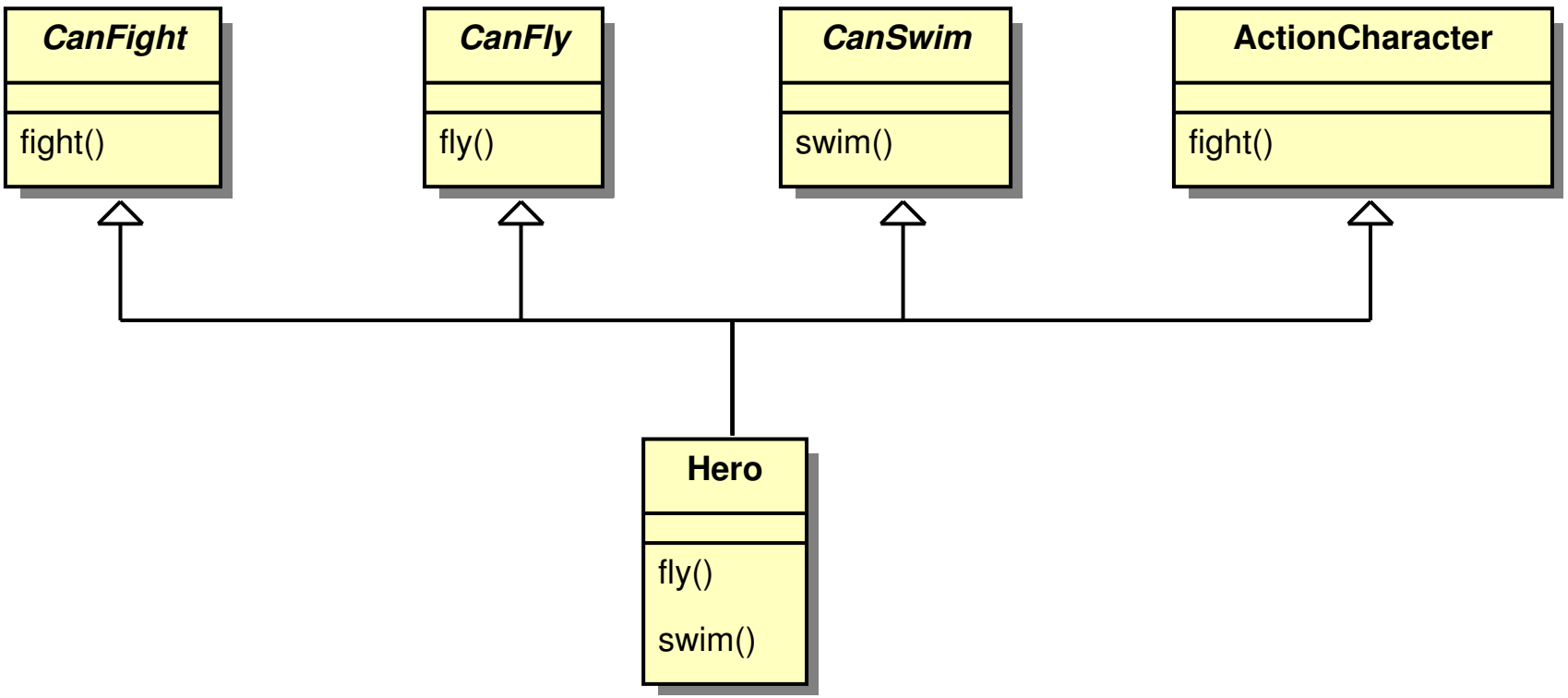
- A class can implement multiple interfaces
  - It makes sense, because sometimes an object can be seen as two different types, depending on the context
- However, a class can extend only one other class
  - The **extend** keyword must precede the **implement** keyword
- The implementation of the interface methods need not to be in the class itself
  - It can be in the base class, or in the derived classes (in the latter case, the class becomes abstract)

Deriving from multiple interfaces

Adventure.java

```
interface CanFight {
    void fight();
}

interface CanSwim {
    void swim();
}

interface CanFly {
    void fly();
}

class ActionCharacter {
    public void fight() {}
}

class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
    public void swim() {}
    public void fly() {}
}
```

UML class diagram



Extending interfaces

- It is possible to extend and combine interfaces to obtain more complex ones
  - To extend an interface with additional methods, you can use keyword **extends**, just as inheriting a derived class from a base class
  - Unlike classes (where you can extend from one class only), you can extend an interface from multiple base interfaces

```
public interface BInt1 {
  void fun1();
  int g();
}

interface BInt2 {
  void fun2();
  int h();
}

interface Der extends BInt1, BInt2 {
  void fun3();
}
```

Outline

Exceptions

- An **exceptional condition** is a run-time error that prevents the continuation of the function or scope the program is currently executing
  - A normal error is something you can deal with in the current scope
  - An exceptional condition is something you cannot do anything about at the current point of the program
  - All you can do is to return from the method, signalling to the higher levels that the method failed to complete
- The typical way to handle this exceptional condition in C is to return an error code
  - For example, many OS functions return a special error code to mean that the function failed to complete successfully
- Consider opening a file for reading. The user specifies the file name, but what happens if a file with that name cannot be found on the disk?
  - The **open()** function cannot do anything about it; it returns a special error code, and the user has to check the return value to see if the function was successful

Exceptions in object oriented

- In procedural languages like C, handling exceptions is annoying
  - For each function, you have to check the return value, and write some code to decide what to do
  - Sometimes, the error cannot be addressed directly, so we have to propagate the error condition to the upper layers
  - This adds a substantial amount of effort for the programmer, and makes the code less readable
- In Object Oriented languages, the idea is that you can handle the error where it is more appropriate through the concept of Exceptions
  - An exception is just an object of a class (everything is an object) that the programmer can **throw** to signal the error condition
  - The exception can then be **caught** where is more appropriate

Syntax

- First, let's use existing exceptions, provided by the Java library

```
MyClass t;
... // some code that works with t
if(t == null)
  throw new NullPointerException();
```

- In this example, we check if reference t points to an object
- If not, we *throw* an exception of type **NullPointerException()**
- Actually, the previous code is superfluous, since the same exception is thrown automatically whenever you use a null reference

Catching exceptions

- To be able to catch an exception, you must enclose the code in a **try** block

```
try {
// code that can throw an exception
} catch (Type1 exc1) {
  // code to handle exceptions of Type1
} catch (Type2 exc2) {
  // code to handle exceptions of Type2
} catch (Type3 exc3) {
  // code to handle exceptions of Type3
}
```

- When an exception is thrown,
  - The JVM looks for handler that catch the exception type
  - If it does not find them, it goes up one scoping level, again searching for the right **catch** block
  - If it does not find it, the program is terminated with an error
- An example:
  `./examples/06.java-examples/SimpleExceptionDemo.jav`

Constructors for Exceptions

- Exceptions are objects, they can be made as complex as we need

FullConstructors.java

```
class MyException extends Exception {
    public MyException() {}
    public MyException(String msg) { super(msg); }
}
```

FullConstructors.java

```
    public static void g() throws MyException {
        System.out.println("Throwing MyException from g()");
        throw new MyException("Originated in g()");
    }
```

FullConstructors.java

```
        try {
            f();
        } catch(MyException e) {
            e.printStackTrace();
        }
```

Throwable

- Your exception must derive from an existing library class called **Exception**, which extends class **Throwable**
  - One of the methods of **Throwable** is **printStackTrace()**
- Here is the complete documentation for Exception

  `http://download.oracle.com/javase/1.6.0/docs/api/java/lang/Exc`

Exception specification

- If your method can throw an exception, you must specify this in the method declaration
  - You must use the keyword **throws** followed by a comma-separate list of exceptions
  - If you do not specify anything, it means your method does not throw anything
  - In Java you cannot lie: if you throw an exception, you have to declare it in the method
- The exception is part of the method signature
  - If a method of a base class specifies an exception list, all derived classes that override that method must specify the same exception list, else the compiler will complain
- Java uses **checked exceptions** (i.e. the check is done at compile time)
  - C++ is more liberal in this respect

Exceptions hierarchy

- As you have already seen, exceptions can be organised into a hierarchy of base-derived classes
- When catching an exception, the normal type rules are applied
  - If you catch **Exception**, you are actually catching any type of exceptions (as they all derive from class **Exception**).

```
try {
  ...
} catch (Exception e) {
  // this catches any type of exception
}
```

- The catch clauses are examined in the same order they are written
  - It is convenient to put the above code at the end of the catch list, so that you first try to catch more specific exceptions, and then more generic ones.

Rethrowing an exception

- Sometimes, when an exception is thrown, it is useful to catch it to do some cleanup, and then throw it again so that the upper layers can handle it
- Therefore, inside a catch clause you can *rethrow* the same exception you have just caught

```
catch(Exception e) {
  System.err.println("An exception was thrown");
  throw e;
}
```

- You can also throw a different exception, of course (that is always possible from anywhere)

Runtime exceptions

- There is an entire hierarchy of special exceptions, whose base class is **RuntimeException**
  - These exceptions are automatically thrown by the JVM
  - An example is **NullPointerException**
  - Another one is **ArrayIndexOutOfBoundsException**
  - It is not necessary to specify these exceptions in the exception specification list of a method (since these ones can originate from anywhere)
  - We say that these exceptions are *unchecked*, because the compiler does not check from them

finally

- After the **catch** clauses, you can insert a **finally** clauses, a block of code that is always executed at the end
  - the **finally** clause is executed when the exception is thrown, and when it is not thrown

```
try {
  // The guarded region: Dangerous activities
  // that might throw A, B, or C
} catch(A a1) {
  // Handler for situation A
} catch(B b1) {
  // Handler for situation B
} catch(C c1) {
  // Handler for situation C
} finally {
  // Activities that happen every time
}
```

- An example:
  `./examples/06.java-examples/AlwaysFinally.java`

Lost exceptions

- Unfortunately exceptions can get lost
  - For example, when inside a finally clause, you call a method that can throw another exception: the second one will overwrite the first one.
- `./examples/06.java-examples/LostMessage.java`