

BALLWORLD, INHERITANCE-BASED*

Stephen Wong
Dung Nguyen

This work is produced by OpenStax-CNX and licensed under the
Creative Commons Attribution License 1.0[†]

Abstract

The module uses the Ballworld program to demonstrate key ideas in abstract classes, polymorphism, inheritance and other coding techniques. This module will focus on an inheritance-based architecture.

In this module we will explore many OOP concepts by examining the program "Ballworld". Download the code for Ballworld here[‡] (**Link Temporarily Disabled**. Please contact the authors for the code.).

*Version 1.8: Sep 6, 2010 2:02 pm -0500

[†]<http://creativecommons.org/licenses/by/1.0>

[‡]See the file at <http://cnx.org/content/m11806/latest/>

UML class diagram of Ballworld

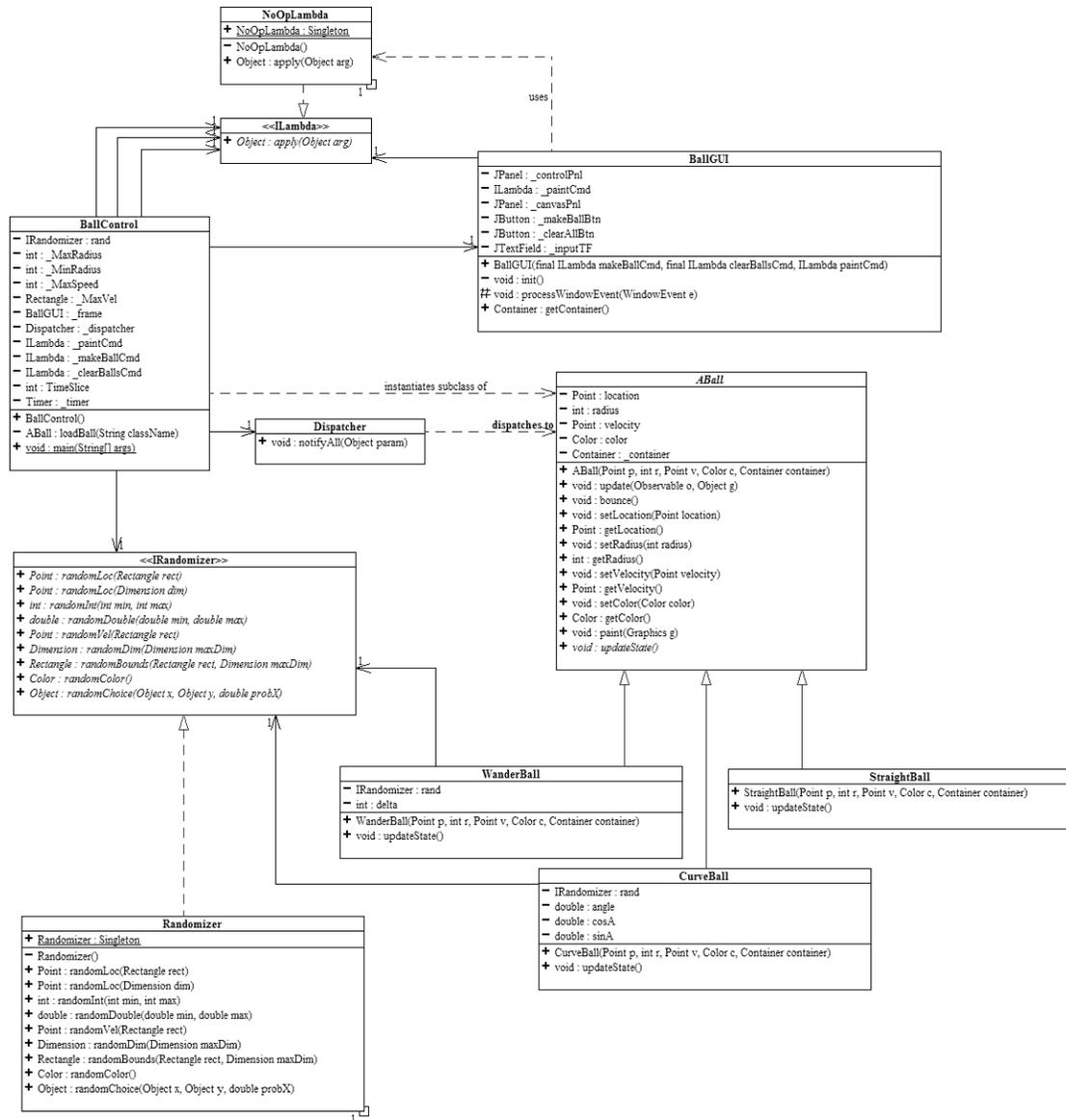


Figure 1: Note the union design pattern w.r.t. ABall and its subclasses.

To run Ballworld, load the files into DrJava and right-click the BallControl file. Select "Run Document's Main Method" to run the program. From the command line, go to the uppermost Ballworld directory and compile all the files in both the directories ("javac ballworld/*.java" and "javac command/*.java") and then run the program as such: "java ballworld.BallControl". The Make Ball button creates an instance of whatever ABall class is typed into the text field on the left. The Clear All button clears any balls off of the screen.

In a nutshell, the way Ballworld works is akin to a flip-book animation: The `BallControl` class contains a `Timer` that "ticks" every 50 milliseconds. Every time the timer ticks, the panel in the `BallGUI` upon which the balls are to be drawn is requested to repaint itself. When the panel repaints, it also tells the `Dispatcher` to notify every `ABall` in the system to update itself. When an `ABall` updates, it updates any internal values it has (its "state") such as its color, radius, and/or velocity. It then moves its position by an amount corresponding to its velocity and draws ("paints") itself onto the panel. Since this is happening 20 times a second, the balls appear to be moving and/or changing and thus the animation is achieved. The `ILambda` interface³ enables the `BallGUI` to communicate in a generic, decoupled manner to the `BallControl` and the `Randomizer` class is a utility class that provides methods to produce various random values needed by the system. Much of the code in Ballworld is significantly more sophisticated than what has been covered in the course so far—it will be covered soon, don't worry!

1 Abstract Classes

First, we will focus on the union design pattern between `ABall`, `WanderBall`, `CurveBall` and `StraightBall`. In a union design pattern, we see that the superclass represents an abstraction of the union of its subclasses. For instance, a fruit is an abstraction of specific concrete classes such as apple and pear. A fruit embodies the common characteristics of its subclasses even if it doesn't completely describe the exact nature of those characteristics. In a fruit, there is a seed. However, the notion of "fruit" doesn't specify exactly the number, size, color or shape of its seed(s). It only specifies that it does indeed have a seed. Likewise, a fruit has the behavior of ripening. Apples, oranges, and bananas all ripen differently and at different rates. The abstract fruit notion does not specify the specific nature of the ripening behavior, just simply that it does have that behavior. In such, we see that we can never have a fruit that is not a specific class of fruit, such as an orange or grape.

Corresponding to the above notions, abstract classes in Java cannot be instantiated. Abstract classes are denoted by the `abstract` keyword in the class definition:

```
public abstract class AFruit {...}
```

By convention, the classnames of abstract classes always begin with "A".

In Ballworld, `ABall` is an abstract class representing a circular ball that has a number of properties: a color, a position, a velocity, etc. The abstract ball also has some defining behaviors, such as that all balls should paint a filled, colored circle when requested to display themselves on a graphics context (a panel). Likewise all balls know how to bounce off the walls of the container. These concrete behaviors are called "default behaviors" because all subclasses of `ABall`, such as `StraightBall` and `CurveBall`, automatically get these behaviors by default. One of the most common and useful reasons for using an abstract class is to be able to define the default behaviors for all the subclasses.

1.1 Abstract Methods

But what about the abstract behaviors that abstract classes exhibit? For instance the abstract "ripening" behavior of a fruit? At the abstraction level of a fruit, the exact implementation of ripening cannot be specified because it varies from one concrete subclass to another. In Java, this is represented by using the keyword `abstract` as part of the signature of a method which has no code body:

```
public abstract class AFruit {  
    // rest of the code  
  
    public abstract void ripen();  
}
```

There is no code body because it cannot be specified at this abstraction level. All that the above code says is that the method does exist in all `AFruit`. The specific implementation of method is left to the subclasses, where the method is declared identically except for the lack of the `abstract` keyword:

```
public class Mango extends AFruit {
// rest of code

public void ripen() {
// code to ripen a mango goes here
}
}

public class Tomato extends AFruit {
// rest of code

public void ripen() {
// code to ripen a tomato goes here
}
}
```

The technical term for this process is called **overriding**. We say that the concrete methods in the subclasses **override** the abstract method in the superclass.

Note that if a class has an abstract method, the class itself must be declared **abstract**. This simply because the lack of code in the abstract method means that the class cannot be instantiated, or perhaps more importantly, it says that in order for a class to represent abstract behavior, the class itself must represent an abstract notion.

Overriding is not limited to abstract methods. One can override any concrete method not declared with the **final** keyword. We will tend to avoid this technique however, as the changing of behavior as one changes abstraction levels leads to very unclear semantics of what the classes are actually doing.

In Ballworld we see the abstract method `updateState`. Abstract methods and classes are denoted in UML diagrams by italic lettering. This method is called by the `update` method as part of the invariant process of updating the condition of the ball every 50 milliseconds. The update method does 4 things:

1. Update the state of the ball by calling the abstract `updateState` method.
2. Move (translate) the position of the ball by adding the velocity to it.
3. Check if the ball needs to bound off a wall.
4. Paint the ball up on the screen.

This technique of calling an abstract method from inside of a concrete method is called the **template method design pattern**—which we will get to later in the course.

`ABall.updateState()` is abstract because at the abstraction level of `ABall`, one only knows that the ball will definitely do something with regards to modifying (perhaps) its internal field values(its "state"). Each subclass will do it differently however. The `StraightBall` will do nothing in its `updateState` method because a ball with a constant (unchanging) velocity will travel in a straight line. Remember, **doing nothing is doing something!** The `CurveBall`'s `updateState` method uses sines and cosines to turn the velocity by a fixed (though randomly chosen) angle at every update event. You can imagine that other possible subclasses could do things such as randomly change the velocity or change the radius of the ball or change the color of the ball.

There is no code in the entire Ballworld system that explicitly references any of the concrete `ABall` subclasses. All of the code runs at the level of abstraction of an abstract ball. The differences in behavior of the various balls made on the screen using the different concrete subclasses is strictly due to **polymorphism**.

New types of balls can be added to the system without recompiling **any** of the existing code. In fact, new types of balls can be added without even stopping the Ballworld program!

2 Abstract classes vs. Interfaces

Subclasses have a different relationship between interfaces and abstract superclasses. A subclass that implements an interface is saying simply that it "acts like" that specified by the interface. The class makes no statements however about fundamentally what it actually is. An actor implements a fierce alien from a distant planet in one movie and a fickle feline in another. But an actor is actually neither. Just because the actor portrayed an interplanetary alien, doesn't mean that the actor fundamentally possessed all the abilities of such an alien. All it says is that in so far the context in which the actor was utilized as the alien, the actor did implement all the necessary behaviors of the alien.

A subclass is fundamentally an example of its superclass. A subclass automatically contains all the behaviors of its superclass because it fundamentally **is** the superclass. The subclass doesn't have to implement the behaviors of its superclass, it already has them. An actor is a human and by that right, automatically possesses all that which makes a human: one head, two arms, 10 toes, etc. Note that this is true even if the abstract class has 100% abstract methods—it still enforces a strict taxonomical hierarchy.

*implements is about **behaving**, extends is about **being**.*

3 Variant vs. Invariant Behaviors

A crucial observation is that the the Ballworld code that manages the GUI (`BallGUI`) and the ball management (`BallControl`, `Dispatcher`, etc.) only deal with the abstract ball, `ABall`. That is, they represent **invariant** behavior at the abstract ball level. The display, creation and management of the balls is independent of the particular kinds of concrete balls that is being handled. The main Ballworld framework can thus handle **any** type of `ABall`, past, present and future.

`StraightBall`, `CurveBall` and `WanderBall` are thus concrete variants of `ABall`. They represent the **variant** behaviors of the system. Other than in their constructors (which will prove to be a significant point when this inheritance-based model is compared to a more advanced composition-based model), these concrete subclasses only code the abstract variant behavior associated with a ball, namely the `updateState` method. Inheritance gives any instantiation of these classes both the invariant behaviors from the `ABall` superclass plus the variant behaviors from the subclass.

The Ballworld code demonstrates the importance of the concept of **separation of variant and invariant behaviors**.

Clearly and cleanly separating the variant and invariant behaviors in a program is crucial for achieving flexible, extensible, robust and correct program execution.

Where and how to separate the variant and invariant behaviors is arguably the most important design consideration made in writing god software.

4 Java Syntax Issues

4.1 Packages

Packages are way that Java organizes related classes together. Packages are simply directories that contain the related code files. Each class file in a package directory should have the line `package XXX;` at its top, where the `XXX` matches with the directory. name. If neither `public` nor `private` (nor `protected` – i.e. a blank specifier) is used to specify the visibility level of a class or method, then that method can be seen by other members of the package but not by those outside of the package. To use the public classes in a package, the `import myPackage.*;` syntax is used. This tells the Java compiler to allow all the public classes in the `myPackage` directory. Packages can be nested, though each level must be imported separately.

4.2 Static fields and methods

Static fields and methods, denoted by the `static` keyword in their declarations, are fields and methods that can be accessed at a class level, not just an object level. In general these are values or behaviors that one wishes for all instances of a class to have access to. These values and behaviors are necessarily independent of the state of any particular instance. Static fields and methods are often referred to as "**class variables**" and "**class methods**".

An examples of a class variables are `Math.PI` and `Color.BLUE` or `Color.RED`. These are universal values associated with math and color respectively and thus do not need an object instance to be viable. By convention, all static field names are in all capitol letters. A static field is referenced simply by writing the class name followed by a period and then by the field name. No instantiations are necessary.

The `Randomizer` class contains numerous static methods. This is because each of the methods to produce various random values is independent of each other and that the process in each method does not affect nor is affected by the state of the rest of the class. Essentially, this entails that the class contain no non-static fields. A class as such is referred to as being "**stateless**". Just like a static field, a static method is invoked in the same manner as the static fields: `ClassName.staticMethodName(...)` Classes with static methods are usually utility classes that are used to hold a set of related functional processes, e.g. `Randomizer` holds a collection of random value generators. Likewise, `Math` holds a combination of static values, such as `PI` and static methods such as `sin()` and `cos()`.

There is one very special static method with the following **exact** signature:

```
public static void main(String[] args)
```

This method, found in `BallControl`, is the method that Java uses to start programs up. Since OO programs are systems of interacting objects, this static "main" method is used to create the first object(s) and get the program up and running. So when Java starts a program, it looks for this and only this method.

4.3 Calling methods of the superclass

When concrete methods or the constructor of a superclass are overridden, sometimes it is necessary or desirable to call the original superclass behavior from the subclass. A common reason for this is that the subclass's behavior is simple an addition to the superclass behavior. One does not want to replicate the superclass code, so a call to the superclass's original methods is required at some point in the subclasses overriding method. To accomplish this, Java uses the `super` keyword. `super` refers to the superclass instance, just as `this` refers to the class instance itself (the subclass here). Note that technically, `super` and `this` are the same object – think of it as the difference between the *id* and the *ego*. (Does that mean that a coding mistake wiith respect to `super` and `this` is a Freudian slip?)

Suppose the superclass has a method called `myMethod()` which the subclass overrides. For the subclass to call the superclass's `myMethod`, it simply needs to say `super.myMethod()`. Contrast this to the subclass calling its own `myMethod`: `this.myMethod()` (note: Java syntax rules allow for the `this` to be omitted).

To make a call to the superclass's constructor the subclass simply says `super(...)`, supplying whatever parameters the superclass constructor requires. This is a very common scenario as the the subclass almost always needs to superclass to initialize itself before it can perform any additional initializations. Thus the `super(...)` call must be the first line in the subclass's constructor. If the no-parameter constructor of the superclass is required, the call to `super` can be omitted as it will be automatically performed by the Java run-time engine. This of course presumes that the superclass's no-parameter constructor exists, which it does **not** if a parameterized constructor has been declared without explicitly declaring the no-parameter constructor.