A Brief Introduction to Java for C++ Programmers: Part 2
ENGI 5895: Software Design

Andrew Vardy

Faculty of Engineering & Applied Science
Memorial University of Newfoundland

January 15, 2017
This second set of notes focuses on the following features of Java:

- Packages
- Inheritance
- Abstract Classes and Methods
- Interfaces
To use a class such as ArrayList from the Java API you have three choices:

1. import the class using its full name:
   ```java
   import java.util.ArrayList;
   (This statement must go at the top of your .java file, outside the class)
   ```

2. import the whole package:
   ```java
   import java.util.*;
   ```

3. Utilize the full class name everywhere.
The following code illustrates **import**, the container class **ArrayList**, and one of the primitive wrapper classes, **Integer** (it also introduces **generics**, Java’s equivalent of templates!):

```java
import java.util.ArrayList;
// ALT: import java.util.*;

public class Import {
    public static void main(String[] args) {
        ArrayList<Integer> list =
            new ArrayList<Integer>();

        // list.add(new Integer(10));
        list.add(20); // Shortcut to above form
        list.add(30);

        for (Integer i : list)
            System.out.println(i);
    }
}
```
A package is a set of related classes

All files belonging to the package must be placed in a corresponding directory
e.g. files in package `avardy.package1` must go in `avardy/package1` (relative to the `CLASSPATH` directory)

A class, member data item, or member method is either private, public, protected, or has package access, meaning that it is public within the package:

```
package mypackage;
public class X {
    private int i;
    int j;
}
```

`j` is accessible from other classes within the package, but not `i`
package avardy.package1;

class A {
    int value = 42;
}

package avardy.package1;

class B {
    private int value;
    public B(A refA) {
        // This is OK because A's
        // value has package access
        value = refA.value;
    }
}

package avardy.package1;

public class Front {
    public static void main(String[] args) {
        A refA = new A();
        B refB = new B(refA);
    }
}
Inheritance in Java is quite similar to C++ with a few exceptions:

- No multiple inheritance
- Singly-rooted hierarchy (all classes inherit from Object)
- Syntax
  - C++: class Derived : public Base
  - Java: class Derived extends Base
- Utilize super keyword to call the base class constructor or base class methods
class Animal {
    protected int legs;

    public Animal(int legs) {
        this.legs = legs;
    }

    public void makeSound() {
        System.out.println("???");
    }

    public String getClassification() {
        if (legs == 2)
            return "biped";
        else if (legs == 4)
            return "quadroped";
        else
            return "unclassified";
    }
}
public class Dog extends Animal {
    private String name, owner;

    public Dog(String name, String owner) {
        super(4);
        this.name = name;
        this.owner = owner;
    }

    @Override public void makeSound() {
        System.out.println("Woof!");
    }

    public static void main(String[] args) {
        Dog dog = new Dog("Bruno", "Andrew");
        System.out.println("Classification: " +
                          dog.getClassification());
        dog.makeSound();
    }
}
Abstract Methods and Classes

In C++ we have the notion of pure virtual methods:

- They have no implementation in the base class, but must be implemented by the sub-classes

In Java, these methods are declared as **abstract**

- A class defined with any abstract methods must be declared as **abstract**
- You cannot instantiate an abstract class! Only a sub-class.
- An abstract class may have implementations for non-abstract methods
abstract class Instrument {
    public abstract void play();
    public String getName() {
        return "Instrument, but you’ll " + "never see this!";
    }
}

class Drum extends Instrument {
    public void play() {
        System.out.println("Bang!");
    }

    public String getName() {
        return "Drum";
    }
}
After adding a Guitar class, we can see that Instrument serves to standardize the interface to sub-classes:

```java
public class TestInstruments {
    public static void main(String[] args) {
        Instrument[] trio = new Instrument[3];
        trio[0] = new Drum();
        trio[1] = new Guitar();
        trio[2] = new Guitar();

        // Usage code is independent of
        // the creation code above.
        for (Instrument inst : trio)
            inst.play();
    }
}
```
Java goes further than abstract classes. An abstract class might contain some implementation:

```java
abstract class Instrument {
    public abstract void play();
    public String getName() {
        return "Instrument, but you'll never see this!";
    }
}
```

But often what we really want is to define the methods that a set of classes must have, **and nothing more**. For this purpose, we have **interfaces** which have no implementation and public access for all fields.

```java
interface Instrument {
    void play();
    String getName();
}
```
Classes can **implement** an interface.

class Drum implements Instrument {
    public void play() {
        System.out.println("Bang!");
    }

    public String getName() {
        return "Drum";
    }
}

Andrew Vardy  Brief Intro. to Java: Part 2
Implementing Multiple Interfaces

Some entities can be interacted with in several different ways. For example, if you have a vehicle you should be able to drive it and check how much gas is left. Some entities may be capable of being repaired.

```java
interface Vehicle {
    void drive(double km);
    double gasLeft();
}

interface Repairable {
    boolean canRepair();
    void repair();
}
```

- A boat is a vehicle
- An alien spaceship might be a vehicle but is probably not repairable
- A toaster is repairable but is not a vehicle.
- A car is both a vehicle and repairable...
class Car implements Vehicle, Repairable {
    double mileage = 0;
    double gas = 100.0;

    // @Override public void drive(double km) {
    //     mileage += km;
    //     gas -= km / 10.0;
    //     // Not handling running out of gas!
    // }

    public double gasLeft() {
        return gas;
    }

    public boolean canRepair() {
        return (mileage < 200000);
    }

    public void repair() {
        System.out.println("Good as new!");
    }

    public double getMileage() {
        return mileage;
    }
}
Features Not Covered

- **Tools outside the Java language itself:**
  - Annotations (e.g. @Override or @Test placed in front of a method)
  - Javadoc: Generate API documentation for your code
  - JAR files: Collections of .class files (and data files)

- **The `final` keyword**
  - Constants:
    - `public static final double LIGHTSPEED = 299792458.0;`
  - Various other uses

- **Inner classes**
- **Exception handling**
- **We saw only a tiny fraction of the Java API!**
- **See links page for more information on these topics**