Organization, Collections, and RTTI

Dr. Isaac Griffith
Idaho State University

Outcomes

After today's lecture you will be able to:

- Understand the basics features of most OO Languages
  - Class Organization
  - Collections
  - Run-time Type Identification

Class Organization

CS 2263

ROAR

Organizing Classes

- In large systems it is essential that components be located in a manner to facilitate easy access
- Since classes and interfaces comprise systems, we must have a method for organizing them
- In java, this method is based on files and packages

Creating Files

- There are several general rules and conventions for file organization
  - One class/interface per file
  - Files are named `<class/interface name>.java`
  - For a class with multiple types
    - Only one outer type may be public, and it is the one the file is named after

ROAR

Packages

- A package is simply a collection of classes
  - In java, they are formed from a physical directory structure
- Packages provide a logical namespace by within which type names must be unique
  - Thus, a class' full name is the `packagename.type_name`
- Package names are the names of directories where the "/" or "\" directory separator is replaced with a period
  - E.g., the directory `edu/isu/cs/` would become the package `edu.isu.cs`
  - Package names tend to start with an inversion of the companies url - the prior example would be the package for "http://cs.isu.edu" the page for the CS Department.
- Examples of common packages from java include:
  - `java.io`
  - `java.util`
  - `java.lang.reflect`

Using Packages

Idaho State University | Computer Science

- All `.java` files must declare the package they belong to as the first executable line of code.
  - This is done with with the `package` declaration:
  - E.g., `package edu.isu.cs;`

- In order to utilize a type from a different package than the current type you have three choices
  1. You can access the type using its full name
     - E.g., `java.util.Vector myVector = new java.util.Vector();`
  2. You can import the type directly
     - E.g., `import java.util.Vector;`
  3. If you are using several types from the same package you can import all types
     - E.g., `import java.util.*;`

ROAR

Encapsulation

- One of the most important features of OOP is that it facilitates *encapsulation* – a class encapsulates both the data it uses, and the methods to manipulate the data

- The external user *only* sees the public methods of the class, and interacts with the objects of that class purely by calling those methods

- This has several benefits
  - Users are insulated from needing to learn details outside their scope of competence
  - Programmers can alter or improve the implementation without affecting any client code

Access Restrictions

Idaho State University | Computer Science

- Encapsulation is enforced by the correct use of the access modifiers, `public`, `private`, and `protected`
- If you omit the access modifier, then you get the default, sometimes known as “package”
- These latter two modifiers are only really relevant for multi-package programs that use inheritance, so we need only consider public and private at the moment

`public` and `private`

- If an **instance variable** is `public`, then
  - Any object can *access* it directly
  - Any object can *alter* it directly
- If an **instance variable** is `private`, then
  - Objects that belong to *the same class* can access and alter it
  - Notice that privacy is a per-class attribute not per-object
- If a **method** is `public`, then
  - Any object can call that method
- If a **method** is `private`, then
  - Objects that belong to *the same class* can call it

Public Methods

- The *public interface* of a class is its list of public methods, which details all of the services that this class provides
- Once a class is released (for example, as part of a library), then it is impossible or very difficult to change its public interface, because client code may use any of the public methods
- Public methods must be precisely documented and robust to incorrect input and accidental misuse
- Classes should make as *few* methods public as possible – limit them to just the methods needed for the class to perform its stated function.

Public Variables

- Normally instance variables should **not** be public, since if client code can alter the values of instance variables then the benefit of encapsulation is lost

- If client access to instance variables is desirable, then it should be provided by *accessor* and/or *mutator* methods (getters and setters)

- Advantages
  - Maintenance of object integrity
  - Permits change of implementation

Simple Example

Idaho State University | Computer Science

```java
class MyDate {
    public int day;
    public String month;
    public int year;
}

MyDate md = new MyDate();
md.day = 31;
md.month = "Feb";
```

Here `md`, is corrupt (since there is no Feb. 31) which could cause problems elsewhere in the system.

ROAR

Use Mutators Instead


```java
public void setDay(int day) {
    // Check that day is valid for this.month
    // before setting the variables
}

public int getDay() {
    return this.day;
}
```

- Setter methods act as “gatekeepers” to protect the integrity of objects.
- Setters reject values that would create a corrupt object.
- Getters return a value for client code to use, but do not allow the object itself to be changed.

Java Module System

CS 2263

ROAR

Java Module Basics

- Added in Java 9
- Modules may contain one or more Packages
- Each module must be given a unique name
  - `edu.isu.cs2263.intro`
  - do not use underscores for any names in Java

Module Root Directory

- In prior versions of java, all packages and classes were packaged under the root dir:
  - a class: `edu.isu.cs2263.intro.App`
  - is in directory: `/edu/isu/cs2263/intro/`
- However modules allow us to package all module components under a directory with the same name as the module:
  - For module: `edu.isu.cs2263.intro`
  - a class: `edu.isu.cs2263.intro.App`
  - is in directory: `/edu.isu.cs2263.intro/edu/isu/cs2263/intro/`
- Furthermore, you should have separate gradle projects for each module
  - Thus having separate `src/main/java` directories

ROAR

Module Descriptor

- A module's definition is stored in a module descriptor file
- A module descriptor file is a `module-info.java` file
  - must be located in the corresponding module's root directory

Syntax:

```
module Identifier {

}
```

- Where `Identifier` is the name of the module.

Example:

```
module edu.isu.cs2263 {

}
```

Module Exports

- A module must explicitly export all packages in the module that are to be accessible for other modules using the module.

- One must also export subpackages
  - but exporting the parent package is not required

Example:

```
module edu.isu.cs2263 {
    exports edu.isu.cs2263;
    exports edu.isu.cs2263.util;
}
```


ROAR

Module Requires

- If a module requires another module to do its work, it must be specified in the descriptor
- This is done with the `requires` keyword

Example:

```
module edu.isu.cs2263 {
    requires javafx.graphics;
}
```


ROAR

What's Not Allowed

Idaho State University | Computer Science

- **Circular Dependencies**
  - You cannot have circular dependencies between packages
  - That is Module A cannot depend on Module B, if Module B already depends on Module A
    - These dependencies may be either direct or indirect
  - In other words, the dependency graph must be acyclic.
- **Split Packages**
  - Only a single module may export a package at any time.
  - Thus, you cannot have two (or more) modules exporting the same package

Module Benefits

The Java Module System provides several benefits.

- Smaller application distributables via the Modular Java Platform
- Encapsulation of internal packages
- Startup detection of missing modules

Additionally, Java will automatically modularize unmodularized dependencies that you use.

Collections

CS 2263

Collections

Idaho State University | Computer Science

- In 2235 and 1187, you (should have) learned all about data structures.
- However, most language base libraries contain a majority of these structures, or there are libraries that will provide them
- The section of java which contains these structures is the Java Collections Library.
  - The primary interface for this library is: `java.util.Collection`, with the following interface
    - `boolean add(Object object)`
    - `boolean addAll(Collection collection)`
    - `void clear()`
    - `boolean contains(Object object)`
    - `int size()`
    - additional methods to remove, check if empty, etc.

ROAR

Collections

Idaho State University | Computer Science

Java provides implementations of several useful collections:

- `List` via the interface `java.util.List`
  - `java.util.ArrayList`
  - `java.util.LinkedList`
- `Stack` via `java.util.Stack`
- `Queue` via the interface `java.util.Queue`
  - `java.util.PriorityQueue`
  - `java.util.Deque`
  - `java.util.ArrayDeque`
- `Set` via the interface `java.util.Set`
  - `java.util.HashSet`
  - `java.util.TreeSet`
- `Map` via the interface `java.util.Map`
  - `java.util.HashMap`
  - `java.util.TreeMap`
- Additionally, I would look into both of the following libraries
  - **Google Guava**
  - **Apache Commons Collections**

Collections Example


```java
import java.util.*;

public class ListUseExample {

    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        for (int i = 1; i <= 10; i++)
            list.add(new String("String " + i));

        for (String s : list)
            System.out.println(s);
    }
}
```

Run-time Type Identification

CS 2263

ROAR

RTTI

Idaho State University | Computer Science

- We need some mechanism that provides the following:
    - Allows us to detect if one class is an instance of another
    - This mechanism must take into account inheritance hierarchies
- We also need the ability to
    - Apply certain functionality to one subclass
    - But not to a sibling subclass
- RTTI solves both of these, and in Java we have
    - RTTI via reflection through the `Class` class in `java.lang`
    - RTTI via the `instanceof` operator

Reflection

- **Reflection** - a mechanism to inspect the Java Runtime and objects via their meta-data
  - Key to reflect in Java is the `Class` class and objects thereof
- An instance of `Class` can be obtained from any non-null object
  - simply call `getClass()` on that object
- `Class` provides several methods of particular interest
  - `getName()` - returns a `String` representation
  - `forName(String)` - static method which returns a Class instance for the named class
  - `getConstructors()` - returns a list of `Constructor` objects which can be used to instantiate an object
  - `getDeclaredFields()` - returns a list of `Field` objects describing the fields declared in the represented class
  - `getDelcaredMethods()` - returns a list of `Method` objects describing the methods declared in the represented class
  - and many others describing all aspects of the class

ROAR

Reflection Example

Idaho State University | Computer Science

```java
Shape shape;

// code to create a Shape object
// and store its reference in shape

if (shape.getClass().getName().equals("Circle")) {
    // take appropriate action
}
```

Unfortunately, the one drawback is that the compiler cannot check whether “Circle” is the proper name or not.

Reflection Example

Thus, we could easily make the following mistake, but it will compile fine

```
Shape shape;

// code to create a Shape object
// and store its reference in shape

if (shape.getClass().getName().equals("circle")) {
  // take appropriate action
}
```

instanceof

- To handle this problem we must use the `instanceof` operator

```
Shape shape;

// code to create a Shape object
// and store its reference in shape

if (shape instanceof Circle) {
  // take appropriate action
}
```

- This operator returns true if shape is an instance of Circle, and false otherwise
  - This also allows the compiler to ensure that such a check can be made

Although useful, code such as this is typically an unwise idea.

Enhancements to `instanceof`

- Since Java 16 you can now use pattern matching in `instanceof`
- This eliminates the need for explicit casts after a type check

Old Way:

```
if (obj instanceof String) {
    String s = (String) obj;
    if (s.length() > 5) {
        System.out.println("> 5 chars");
    }
}
```

New Way:

```
if (obj instanceof String s &&
                 s.length() > 5) {
    System.out.println("> 5 chars");
}
```


ROAR

For Next Time

- Review Chapter 4.1 - 4.4
- Review this Lecture
- Come to class
- Read Chapter 4.6
- Read the Gson Tutorial
- Read the JavaDoc Tutorial
- Start working on Homework 03

Are there any questions?