Java Reference & Data Structures

Japjot Singh

October 23, 2018

1 A Brief Java Reference

1.1 Object-Oriented Mechanisms

1.1.1 Private/Final Class Structure

A class can't be marked as `private`, but a non-`public` class can be subclassed only by classes in the same package as the class, that is, classes within a different package will not be able to subclass. A `final` class means that it is the end of the inheritance. Furthermore, if a class has only `private` constructors, it cannotbe subclassed. If a class is `private` then it cannot be subclassed, it marks the end of an inhertance. If a method is `private` then it cannot be overriden.

1.1.2 Overriding and Overloading

When you override a method from a superclass you must retain identical arguments (order matters), compatible return types, and the method itself cannot be less accessible. Method overloading, is nothing more than having two methods with identical names but different argument lists, Overloading allows for differing return types but requires a change to the argument list, and accomodates varying access in any direction.

1.1.3 The Stack and the Heap

There are two import areas of memory: The Stack and The Heap. The Stack is where all method invocations and local variables will reside, simliar to how frames work, and that leaves The Heap to hold all objects and the instance variables within objects. An important note is that Object reference variables are still primitive variables and as such if they are declared as local variables, they belong on the stack. All objects, regardless of their reference being a local or instance variable, belon on The Heap.

> Instance variables have a **default** value: 0/0.0/false/null

1.1.4 Constructors

It is a good idea to overload constructors in order to have a default no-arg constructor which will initialize the class object with default values. The compiler will o provide a no-arg constructor only if the user doesn't make any constructor at all. In order for a class to contain more the one constructor, the class **must** have **different** argument lists. Constructors can be `public`, `private`, or just default. Having private constructors will prevent anyone outside from the class from callign the constructor.

Constructor chaining is the phenomenon thats occurs as a result of object creation. When an object is created on The Heap, all the constructor's in the object's inheritance lineage must also run first before the new object is made. If the user doesn not provide a call to `super()` inside the constructor of a subclass, the complier will put a no-arg call to `super()` in each of the user's overloaded constructors and if the superclass has overloaded constructors, the one with no-arg will be called.

Making an analogy between inheritance and parenthood helps clarify the fact that the supreclass parts of an object have to be fully-formed before the subclass parts can be constructed. The superclass constructor must finish before it subsclass constructor. Thus the compiler will fail if `super()` is not the first statement in each constructor.

```
public abstract class Professional {
   private String name;

   public String getName() {
      return name;
   }

   public Professional (String myName) {
      name = myName;
   }
}
public class Doctor extends Professional {

   public Doctor (String name) {
      super(name);
   }
}
public class trainDoctor {
   public static void main (String[] args) {
      Doctor d = new Doctor("Phil");
      System.out.println(d.getName());
   }
}
```

Super class contains private instance variable and a public getter method for said instance variable and within constructor it sets that instance variable. Subclass is able to use super(name) to modify that private method within the superclass and then utilize the get method on its own object as the getter is inherited by the subclass.

Use `this()` to call a constructor from another overloaded constructor within in the same class. The call to `this()` can be used only within the the constructor and must be the first statement in the constrcutor, implying no constructor can contain bother a call to `this()` and `super()`.

An object's eligibility for garbage collection:

- Object reference goes out of scope, permanently
- Object reference is assigned to another object
- Object reference is explicitly set to null

1.1.5 Autoboxing

Java will take care of wrapping primitives to their respectable object types and visa-versa, effectively letting the user specify either primitive or object while permitting both. This can be used in method argumetns, return values, boolean expressions, operations on numbers, and assignments. Wrappers are particularly useful because of their static utility methods, allowing for easy conversion between a String and a primitive `Integer.parseInt(s)`, `Boolean("true").booleanValue()`.

1.2 Interfaces and Abstract Classes

Concrete clases are those which are specific enough to be instantiated, others whose sole purpose is to serve as a polymoprhic argument or return type should be declared `abstract`.

1.2.1 Abstract Class and Abstract Methods

The compiler will not allow a user to instantiate an abstract class. An abstract class is virutally useless (unless it has `static` members), unless it is extended.

Methods can also be called abstract. The way an abstract class is written to be *extended*, an abstract method is written to be *overriden*. If the user declares an abstract method, the user MUST also mark the class as abstract. There cannot be an abstract method in a non-abstract class. The user **must** implement all abstract methods. Implementeing an abstract method is similar to overriding a method. All the abstract methods of a lineage of abstract classes must be implemented by the *first* concrete class down the inheritance tree.

1.2.2 Object Reference and Compile Time

If a user makes an ArrayList of Object, when the user accesses the List they will in get type-Object in return.

The compiler decides whether the user can call a method based on the reference type, not the actual object type, once it find the proper method signature and compiles the compiler will proceed to runtime where Java will look for the most specific dynamic type with the same signature as the reference siganture and run that. **p215 helpful image

In order to bypass the object reference issue the user can down cast the object to its specific type and then proceed to treat the object as if it was the specific type all along. If there is uncertainty behind the specific type, the `instanceof` operator allows for a quick check before casting `Dog d = (Dog) o`

At compile time the compilter check that the method being called on an object is defined in the reference type of the object. Java checks the class of the **reference** variable, not the class of the **object** (specific/dynamic type).

1.2.3 Interfaces

A Java Interface provides the polymorphic benefits of inheritance without the issues and complexities that comes with multiple inheritance. A Java interface is just a purely abstract class, where a subclass which `implements` the interface must implement all the abstract methods.

Important Remarks

- Extend only one abstract class, implement multiple interfaces
- Use an abstract class when describing an IS-A relationship, and you won't be instantiating the class, an abstract class can have both abstract and non-abstract methods
- Use an interface when the user wishes to define a role that the other classes can play, regardless of where they lie in an inheritance tree, must have all abstract methods
- (Insert interface use-case)
- To invoke the superclass version of a method from a subclass that's overrident hte method, just use the `super` keyword

1.2.4 Exceptions

Use exceptions to to handle "exceptional situations" at runtime. To indicate that a method throws an exception the user must add a `throws` clause in the method's declaration. A method can throw more than one exception. Using a try/catch block put the potentially exception situation within the `try` clause and the procedure to perform if there is an exception within the `catch` clause. Exceptions have two key methods: `getMessage()` and `printStackTrace()`. These mehtods can be used as declarations within the `catch` clause which takes as parameter the object `Exception` or any subclass thereof (i.e. `catch(IOException ex)`).

The user should always take note of which method will `throw` an exception and which method will `catch` it. When writing code that might throw an exception, the user must *declare* the exception.

```
void exceptionalMethod() throws FooException, IOException{
   if (somethingWentWrong) {
      throw new FooException();
   } else if(someOtherWrong) {
      throw new IOException();
   }
}

void potentialSituation() {
   try {
      obj.exceptionalMethod();
   } catch (FooException fe) {
      System.out.println("neck");
//    fe.printStackTrace();
   } catch (IOException ioe) {
      System.out.println("notpop");
   }
   System.out.println("pop");
}
```

In the code block above if `obj.exceptionalMethod()` throws a `FooException` the console would print `neck` followed by `pop`. If no exception is thrown the console would simply print `pop`.

Another utility with try/catch is the `finally` clause. The `finally` clause always executes:

- **If the try block fails** (an exception), flow control moves to the `catch block`, after the `catch` block completes, the `finally` block runs, after the `finally` block completes the rest of the method continues on
- **If the try block succeeds** (no exception), flow control skips the `catch` block and the `finally` block runs, after the `finally` block completes, the rest of the method continues
- **If the try or catch block has a return statement `finally` will still run**, flow jumps to `finally` and then back to the return

Runtime Exceptions (i.e. NullPointerException, DivideByZero, ClassCastException) are a subclass of Exception, but they are **not** checked by the compiler. They are known as unchecked exceptions. The reason for this is that try/catch is meant to catch exceptional situations, **not** flaws in the user's code.

The user can declare or catch exceptions using the supertype of the exception thrown, but needs to note that although one can technically catch everything with the parent `Exception` it is often wise to write a different catch block for each exception that needs to be handled uniquely.

When putting catch blocks in sequence it is really important that the user sequence them from more specific to less specific, with the exception at the bottom being the one highest up on the inheritance tree. With catch blocks, the JVM simply starts at the first one and works its way down until it finds a catch that's broad enough.

If the user wishes to bypass handling an exception in the current method, they can declare it and pass the exception to the next method down the stack.

```
   void foo() throws SomeException {
      obj.doExceptional();
   }
```

In the example above, if `obj.doExceptional()` throws an exception, the responsibility of handling the `SomeException` will be handed off to the next method down the stack, in this case that would be the method which made the call to `foo()`. If the exception makes it all the way down the stack to `main()` and the exception has not been handled JVM will simply shut down.

Exception Remarks:

- The user **cannot** make a `catch` or `finally` without a `try`

- The user **cannot** put code between the `try` and `catch`
- A `try` **must** be followed by either a `catch` or a `finally`
- A `try` with only a `finally` (no catch) must still declare the exception in the method, meaning the method must contain the `throws` clause

1.2.5 Java API Library

Packages are helpful in maintaining organization, preventing name-scoped collision, and providing security. Every Java library belongs to a package, and in order to use the library the user must import the package. The general structure of an import is `import java.util.ArrayList`, by importing the package the user can avoid writing out the full name when using the library within their class.

An important fact to note is that an `import` statement does not actuallymake the size of a class bigger, it simply saves the user time by not having to type out the entire name of the class the user wishes to use.

1.2.6 Packages: Motivation and Usage

The primary motivation other than simple organization is the fact that by using packages the user has a clean method to circumvent potential class name conflicts. A common package naming convention is to prepend every class with your reverse domain name, somehting that is gauranteed to be unique.

The procedure to put a class in a package is relatively simple. First, choose a unique package name, the convention mentioned above is a great place to start. Next, put a package statement as the first statement in the source code file (e.g. if my package name was

`me.japjotsingh`)

then the first line of the class I wanted to add to the package would be

`package me.japjotsingh`.

Finally, the user must place the class in a matching directory structure. So if we stick with the package stated earlier, the user must place the class they wish to include in the directory

`/project/source/me/japjotsingh/[insert class].java` .

The final hurdle is to compile and run with packages, to do so the `-d`(directory) flag is both reliable and easy to work with. Sticking with the previous example simply cd the source directory and compile

```
javac -d ../classes me/japjotsingh/[insert class].java
```

and then to compile all the .java files simply use

```
javac -d ../classes me/japjotsingh/*.java
```

and run it with

`javame.japjotsingh.[insert class]`
Note: once the user has

made the package they must always use the full name to access the class.
The `-d` flag is particularly useful in that it allows the user to only have to take care of the structure within the source directory as the flag will tell the compiler to make the proper directory under classes by itself, in fact this is good practice as it ensrues the user doesn't have typos which would lead to compile errors.

Making an executable JAR with packages

1. Make sure all class files are within the correct project structure under the classes directory
2. Create a manifest.txt which will state which class has the `main()` method, use the fully-qalified class name, for example

   ```
   Main-Class:  me.japjotsingh.[insert main class]
   ```

3. Run the jar tool to create the JAR file containing package directories and the manifest

   ```
   $ cd Project/classes
   $ jar -cvmf manifest.txt packEx.jar me
   ```

1.2.7 Anonymous and Static Nested Classes

By defining a static class within an outer class the user has the flexibility invoke the static class without interacting with the outer class.

```
1  public class Outer {
2     static class Inner {
3        void pop() {
4           System.out.println("inner pop");
5        }
6     }
7  }
8  class Driver{
9     public static void main(String[] args) {
10       Outer.Inner popper = new Outer.Inner();
11       popper.pop();
12    }
13 }
```

The code above would simply print to the console `inner pop`. Since the static inner class is a member of the outer class it will still get access to any priavte members of the out class *but only the members that are also static*. Since there is no relationship between the static inner and an instance of the outer class, the static inner class ahs no way to access the non-static (instance) variables and methods of the Outer class.

There is a key naming distinction between a *nested* and *inner* class: any Java class defined within another class is considered *nested* but only *non-static* nested classes are referred to as *inner classes*. All inner classes are nested classes, but not all nested classes are inner classes.

Anonymous inner classes

The primary motivation behind these is that the user can create an *anonymous* inner class and instantiate it on spot. This capability allows the user to pass an entire *class* where they would normall pass only an *instance* into a method argument. So in a case where the user may want to create an `ActionListener` on a `JButton` insated of creating a new class that implements the `ActionListener` interface and then passing in a reference to an instance of an inner class the user can simply instantiate it on the fly.

```
...{
JButton button = new JButton("click");
...
button.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent ev) {
      System.exit(0);
      }
   }); ...
}...
```

In the example above by calling `new ActionListener()` the user created a new class (with no name) that implements the `ActionListener` interaface.

1.2.8 Access Levels and Access Modifiers

The four levels are

- *public* - any code anywhere can access this modified class, variable, method, constructor, etc...
- *protected* - works like default, **except** it also allows subclasses outside the package to inherit the proteced modifed thing
- *default* - only code within the same package as the class with the default modified thing can access the default modified thing
- *private* - only code within the same **class** can access the private modified thing, this private modified thing is private to the class, **not** private to the object

Algorithms

2 Introduction

Complexity is a means of measure how long it takes for some operation to complete, measure using O, Ω, Θ, signifiying worst-case, best-case and the intersection of both, respectively.