Language Elements
Part 1

Angel Gruev
Dreamix ltd.

Packages

- **Definition:** A *package* is a grouping of related types providing access protection and name space management

- *types* refers to
  - Classes
  - Interfaces
  - Enumerations
  - Annotations

Why packages?

- You and other programmers can easily determine that these types are related.
- The names of your types won't conflict with the type names in other packages because the package creates a new namespace.
- You can allow types within the package to have unrestricted access to one another yet still restrict access for types outside the package.

Naming Conventions

- Names are written in all lowercase to avoid conflict with the names of classes or interfaces
- Companies use their reversed Internet domain name
- Packages in the Java language itself begin with `java.` or `javax.`

Packages

- To use a public package member from outside its package:
  - Refer to the member by its fully qualified name
  - Import the package member
  - Import the member's entire package
- Hierarchies of Packages – packages are not hierarchical !
- Name Ambiguities
- Default package

Static Imports

- Import the constants and static methods that you want to use
  - do not need to prefix the name of their class
  - `import static java.lang.Math.PI;`
  - `import static java.lang.Math.*;`
  - can result in code that is difficult to read and maintain
- Constants Interface Pattern – do not use it !

```
public interface PhysicalConstants {
  static final double AVOGADROS_NUMBER = 6.02214199e23;
  static final double BOLTZMANN_CONSTANT = 1.3806503e-23;
  static final double ELECTRON_MASS = 9.10938188e-31;
}

public class MyClass implements PhysicalConstants { … }
```

CLASSES AND INTERFACES

Accessibility of classes and members

- Minimize it!
- Well-designed module
  - hides its internal data and other implementation details from other modules
  - Modules then communicate only through their APIs
  - Encapsulation & decoupling
  - Effective performance tuning
- Make each class or member as inaccessible as possible!

Access Control

- Four possible access members
  - **Private**
  - **Package-private (default access)**
  - **Protected**
  - **Public**
- Applied to fields, methods, nested classes, and nested interfaces

Rules

- After designing your public API, make other members **private**
- Often use of default access means poor design
- Protected and Public members must be supported **forever!**
- Instance fields should never be public

Constants

- Using mutable objects for constants

```
// Potential security hole!
public static final Thing[] VALUES = { ...
};
```

- Fix it :

```
private static final Thing[] PV = { ... };

public static final List<Thing> VALUES =
Collections.unmodifiableList(Arrays.asList(PV));
```

  - Or return a copy of the array

Public fields

- In public classes, use accessor methods, not public fields
  - In package-private classes is OK

```
// Degenerate classes like this should
not be public!
class Point {
  public double x;
  public double y;
}
```

- Expose field that is immutable - less harmful

Immutable Objects

- Immutable - instances cannot be modified
  - easier to design, implement, and use
- How to make a class immutable:
  - Don’t provide mutators
  - **Ensure that the class can’t be extended**
  - **Make all fields final**
  - **Make all fields private**
  - **Ensure exclusive access to any mutable components**
- Immutable objects are inherently thread-safe; they require no synchronization
- Not only can you share immutable objects, but you can share their internal
- Examples: String, BigInteger, BigDecimal, Integer, etc.

Immutable Objects [2]

- Immutable objects make great building blocks for other objects
- Disadvantage - require a separate object for each distinct value
  - i.e. you have a million-bit Big-Integer and you want to change its low-order bit
  - Multistep operations
  - Package-private mutable companion class (BigInteger)
  - Public mutable companion (String, StringBuilder)
- How can you forbid a class to be subclassed?
- If a class cannot be made immutable, limit its mutability as much as possible.

Final modificator

- Final classes
  - if its definition is complete and no subclasses are desired or required
  - Both final & abstract ?
- Final methods
  - prevent subclasses from overriding or hiding it
- Final fields
  - Cannot change the reference
  - Must be initialized

Static modifier

- Static classes – only inner classes
- Static methods
  - Class methods
  - Cannot access instance members
  - Both static & abstract ?
- Static Fields
  - Class variable
  - Can be used both in static or non-static context

Other modifiers

- Transient
  - Only on fields
  - Indicates that these fields are not persistent
- Volatile
  - Used for shared variables in multithreading
- Native
  - Used for methods implemented in platform-dependant code (like C++)
- Strictfp
  - used to restrict floating-point calculations to ensure portability

Inner classes

Concept

- Inner classes let you define one class within another
- Four types of nested classes:
  - Regular (inner)
  - Static
  - Method-local
  - Anonymous

Regular

- How to define:

```
class MyOuter {
    class MyInner {…}
}
```

results in:

```
MyOuter.class
MyOuter$MyInner.class
```

Regular

- Inner class has access to outer's instance members

```
class MyOuter {
  private int x = 7;
  // inner class definition
  class MyInner {
    public void seeOuter() {
      System.out.println("Outer x is " + x);
    }
  } // close inner class definition
} // close outer class
```

Instantiating

- Instantiating an Inner Class from Within the Outer Class
  - In an non-static context of the outer class
- Creating an Inner Class Object from Outside the Outer Class Instance Code

```
public static void main(String[] args) {
  MyOuter mo = new MyOuter();
  MyOuter.MyInner inner = mo.new MyInner();
  inner.seeOuter();
}
                    or
MyOuter.MyInner inner = new MyOuter().new MyInner();
```

Referencing

- To reference the inner class instance itself, from *within the inner class code,* use this.
- To reference the "*outer this" (the outer class instance) from within the inner* class code, use NameOfOuterClass.this
  - MyOuter.this

```
class MyInner {
  public void seeOuter() {
    System.out.println("Outer x is " + x);
    System.out.println("Inner class ref is " + this);
    System.out.println("Outer class ref is " + MyOuter.this);
  }
}
```

Method-Local Inner Classes

- You can also define an inner class within a method

```
class MyOuter2 {
  private String x = "Outer2";
  void doStuff() {
    class MyInner {
      public void seeOuter() {
        System.out.println("Outer x is " + x);
      } // close inner class method
    } // close inner class definition

   MyInner mi = new MyInner(); mi.seeOuter();
  } // close outer class method doStuff()
} // close outer class
```

Access to outer class

- The inner class object cannot use the local variables of the method the inner class is in

```
class MyOuter2 {
  private String x = "Outer2";
  void doStuff() {
    String z = "local variable";
      class MyInner {
        public void seeOuter() {
          System.out.println("Outer x is " + x);
          System.out.println("Local variable z is " +
z); // Won't Compile!
        } // close inner class method
      } // close inner class definition
   } // close outer class method doStuff()
} // close outer class
```

- Unless the local variable is final!!

Anonymous inner class

- Sometimes we don’t need an inner class to be named

```
class Popcorn {
  public void pop() {
    System.out.println("popcorn");
  }
}

class Food {
  Popcorn p = new Popcorn() {
    public void pop() {
      System.out.println("anonymous popcorn");
    }
  };
}
```

Anonymous inner class [2]

- We use often anonymous classes for implementing interfaces

```
Runnable r = new Runnable(); //invalid!! This is an interface

Runnable r = new Runnable() { valid!!
  public void run() { }
};
```

- Can be implemented only one interface
- Can be passed as a parameter to a method

Static Nested Classes

- Similar to normal classes, but placed inside other class
  - Its more about namespace resolution rather relation
- Use static modifier to declare it

```
class BigOuter {
   static class Nested { }
}
```

- Instantiating
  - MyOuter.MyInner mi = new MyOuter.MyInner()
- Does not have access to instance variables

Question

Given:

```
1.  public class TestObj {
2.    public static void main(String[] args) {
3.      Object o = new Object() {
4.        public boolean equals(Object obj) {
5.          return true;
6.        }
7.     }
8.     System.out.println(o.equals("Fred"));
9.    }//main method
10. }//class
```

What is the result?

A. An exception occurs at runtime
B. true
C. Fred
D. Compilation fails because of an error on line 3
E. Compilation fails because of an error on line 4
F. Compilation fails because of an error on line 8
G. Compilation fails because of an error on a line other than 3, 4, or 8

Enums

Enumerated Types

- A type whose legal values consist of a fixed set of constants
  - Seasons of the year, planets in solar system, etc
- Before Java 1.5 int constants pattern was used
  - Unfortunatelly, after java 1.5 some developers still use int constants
  - provides nothing in the way of type safety and little in the way of convenience
  - No separate namespaces

Enums

- Declared like classes
  - But using keyword “enum” instead of “class”

```
public enum Season{
        WINTER, SPRING, SUMMER, FALL
}
```

- they are classes that export one instance for each enumeration constant via a public static final field

```
public void goOnVacation(Season s){ … }
```

- Compile time Type safety!

Adding data and methods to enum

- Enum types let you add arbitrary methods and fields and implement interfaces.
- To associate data with enum constants:
  - declare instance fields
  - write a constructor that takes the data and stores it in the fields

```
public enum Planet {
  MERCURY(3.302e+23, 2.439e6),
  VENUS (4.869e+24, 6.052e6),
  EARTH (5.975e+24, 6.378e6),

  private final double mass; // In kilograms
  private final double radius; // In meters

  // Constructor
  Planet(double mass, double radius) { … }
}
```

Implicit methods

- Every enum has implicitly the methods:
- public static E[] values();
  - return an array containing the constants of this enum type, in the order they're declared
- public static E valueOf(String name);
  - Returns the enum constant with the specified name

Enum and methods

- We can add methods to the enum and even more…

```
// Enum type with constant-specific method implementations
public enum Operation {
  PLUS { double apply(double x, double y){return x + y;} },
  MINUS { double apply(double x, double y){return x - y;} },
  TIMES { double apply(double x, double y){return x * y;} },
  DIVIDE { double apply(double x, double y){return x / y;} };

  abstract double apply(double x, double y);
}
```

Enumeration sets

- Traditionally use bit operation to combine enumeration values:

```
public class Text {
  public static final int STYLE_BOLD = 1 << 0; // 1
  public static final int STYLE_ITALIC = 1 << 1; // 2
  public static final int STYLE_UNDERLINE = 1 << 2; // 4
  public static final int STYLE_STRIKETHROUGH = 1 << 3; // 8

  public void applyStyles(int styles) { ... }
}

text.applyStyles(STYLE_BOLD | STYLE_ITALIC);
```

- Again – not type safe, hard to interpret (what does 13 means ?!), hard to iterate

Use EnumSet instead of bit fields

- Here is how the previous example looks when modified to use enums instead of bit fields. It is shorter, clearer, and safer:

```
// EnumSet - a modern replacement for bit fields
public class Text {
  public enum Style { BOLD, ITALIC, UNDERLINE, STRIKETHROUGH }

  // Any Set could be passed in, but EnumSet is clearly best
  public void applyStyles(Set<Style> styles) { ... }
}
```

text.applyStyles(**EnumSet.of(Style.BOLD, Style.ITALIC));**

Extends enum

- while you cannot write an extensible enum type, you can emulate it by writing an interface to go with a basic enum type that implements the interface

DREAMIX
Software Development

Q & A