CP213 Lesson 10

Interfaces

* Like classes, an interface is a type
* It is the object’s action menu
* Specifies a set of methods that any object that implements the interface should have
* Serves a similar function as a base class
* Uses the keyword interface in the definition where “class” would normally be
* Can only contain the following:
  + Constants
  + Method signatures
  + Default methods
  + Static methods
  + Nested types (inner and outer classes)
* Inconsistency will occur if a class implements multiple interfaces that have the same constants or the same methods with different return types

The comparable interface

* Has only the compareTo method
* It is the programmer’s responsibility to follow the meanings of the comparable interface when implementing it. The compareTo method must return
  + A negative number if the calling object “comes before” the parameter other
  + Zero if the calling object “equals” the parameter other
  + A positive number if the calling object “comes after” the parameter other
* If the parameter other is not of the same type as the class being defined, a ClassCastException should be thrown

The Serializable interface

* Contains no methods
* Only purpose is to write an objects state to a file to be saved for future use

The Cloneable interface

* Contains no method headings or defined constants
* Used to indicate how the method clone should be used and redefined
* Object.clone() creates a bit by bit copy of the objects data in memory, works well if the data is all primitive types or immutable class types (like String)

public class YourCloneableClass implements Cloneable {

// ...

public Object clone() {

try {

return super.clone();

} catch (CloneNotSupportedException e) {

return null;

}

}

// ...

}

If the object contains instance variables who’s type is a mutable class:

1. Invoke the clone method of the base class Object (or whatever the base class is)
2. Reset the values of any new instance variables whose types are mutable class types. This second step is done by making copies of the instance variables by invoking *their* clone methods.

This will only work of the Cloneable interface is implemented properly for the classes to which the instance variables belong

public class YourCloneableClass2 implements Cloneable {

private DataClass someVariable;

// ...

public Object clone() {

try {

YourCloneableClass2 copy = (YourCloneableClass2) super.clone();

copy.someVariable = (DataClass)someVariable.clone();

return copy;

} catch (CloneNotSupportedException e) {

return null; // To keep the compiler happy.

}

}

// ...

}

Default methods

* If for some reason you want to change the interface to add a new method, add the keyword default in front of it so you don’t have to change every class that implements the interface as well.

Defined constants in interfaces

* Any variable defined in an interface can be access from classes that implement it, essentially making it static
* These variables also cannot be changed which makes them final as well as public

Depending on your processor (32 bit vs 64 bit), division will have a different result.

* For this reason Java uses strictfp which restricts floating point calculators so they all have the same result

If your code uses a method that belongs to another programming language you have to use the keyword native

Inner classes

* Classes defined within another class (the outer class)
* Advantages of inner classes:
  + They can make the outer class more self-contained since they are defined inside a class
  + Both of their methods have access to each other’s private methods and instance variables
* Usually used as a helper class

Ex. This is all allowed public class ClassWithInnerClass {

private int outerX;

private String outerY;

private void doSomething () {}

class Inner {

public int innerX;

private String innerY;

public void innerTest() {

doSomething() ;

System.out.println(outerX);

}

public void callOuterMethod(Inner inObject) {

String s = inObject.innerY;

outerTest();

innerTest();

}

}

* When the code is compiled it will have multiple .class file extensions separated by a $
* Inner classes must be static if
  + the object of the inner class is created within a static method of the outer class
  + the inner class must have static members ex. Static int x
* If an inner class is marked public it can be used outside the outer class
* If the inner class is not static it must be created using an object of the outer class

Ex. Account is outer, money is inner

BankAccount account = new BankAccount();

BankAccount.Money amount = account.new Money("41.99");

* Amount can now reference methods from the inner class but not the outer class

Ex. If the inner class is static

BankAccount.Money myAccount = new BankAccount.Money("45.2") ;

The generic type

public class Material <T> {

// T stands for "Type"

private T mtype ;

public void set(T mtype) {this.mtype = mtype ;}

public T get() { return mtype; }

}

* T stands for type and ensures that the same object is used
* You can pass any object except the primitive ones with it
* The extensively used type parameter names are E, K, N, T, V, S, and U.  E (elements) used with the collections, K used for keys, N for numbers, T for types, V for values and S and U for 2nd and 3rd types.
* .class for inner class
  + When compiled, a .class file extension will be created for the outer class as well as all inner classes, with the inner class one having the outer class name, followed by a $
* Cloneable interface
  + Contains no method headings or defined constants. It is used to indicate how the method clone should be used and redefined. Object.clone() creates a bit by bit copy of the objects data in memory, works well if the data is all primitive types or immutable class types (like String)
* Comparable interface
* Default
* Default method
* Inner classes
  + An inner class is a class defined within an outer class, it can access all public and private identifiers of the outer class and vice versa
* Interface
  + Like classes, an interface is a type. It is the object’s action menu that specifies a set of methods that any object that implements the interface should have. Hence, it serves a similar function as a base class
* Native method
* Outer classes
* Serializable interface
* Strictfp
  + Depending on your processor (32 bit vs 64 bit), division will have a different result. For this reason Java uses strictfp which restricts floating point calculators so they all have the same result
* type parameters