7

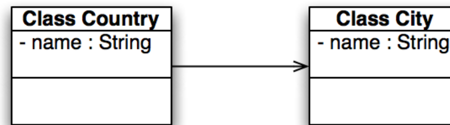
Class Relationships

Self-review Questions

7.1 *How is association between classes implemented*?

An association between two classes is realized as a link between instance objects of the classes at runtime. With Java the link is represented by an object reference. An association represents a long-term relationship between instance objects, so the reference is stored in an instance variable.

In this example:



class Country has a uni-directional association with **class** City, so a Java Country object has a reference to a City object. The reference will be stored in an instance variable belonging to Country. Note, however, that the Country class in the UML diagram fragment does not show an instance variable (attribute) in the icon as the need for the variable is denoted by the association line. Putting in both the variable and association would be redundant. This does mean that you have to take into account associations when determining what instance variables a Java class should have and should not rely on just what is shown in the icon.

7.2 *When is association more appropriate to use than inheritance*?

Inheritance should be used only when a subclass object can be used or substituted where the superclass type was specified. This means that the subclass must meet the entire behavioural specification of the superclass either by inheriting methods or overriding methods to behave in a compatible way. In addition, the subclass must have an *is-kind-of* relationship with the superclass so that all the inherited behaviour is relevant to the subclass. A subclass must never be written to support only part of the behaviour of a superclass.

If a class needs to use only some of the features of another class, inheritance should not be to gain those features. Instead the class should use an association implemented via a private instance variable.

The classic example of the misuse of inheritance is trying to implement a Stack class by subclassing a collection class like Arraylist. A stack has a different set of public methods from an

Arraylist (for example, push, pop and top). Inheritance, however, would give the stack all the public methods of Arraylist, which would be invalid as many of the methods are not part of the stack abstraction. Instead, a Stack class should use an Arraylist by private association.

7.3 *What does object ownership mean?*

An object is owned when it is controlled by another object and accessible only by that other object. Typically this occurs when a class defines a private instance variable (or association with UML) used to reference an object that forms part of the implementation of a class instance and is not intended to be accessible from outside the scope of the class. For example, a Stack class can have a private instance variable of type Arraylist to provide an object to hold the stack data. The Arraylist is never accessible from outside the Stack class and is completely controlled by the Stack class.

Ownership also implies control of the lifetime of the owned object. The owning object creates the owned object and determines when it becomes unused and inaccessible.

7.4 *How is a subclass object initialized?*

Once the JVM has created the space for the instance, the superclass part of the instance is initialized then the explicitly initialized fields are initialized and all instance initializers executed, then the constructor executed. Searching up the inheritance hierarchy eventually stops because superclass is Object which has no superclass.

7.5 *What are the advantages and disadvantages of using protected variables?*

In object-oriented programming generally, protected variables are variables are ones that can be manipulated directly by the class and all subclasses. The advantage is that it makes it easier for subclasses to work with the whole of the state. The disadvantage is that the subclass code needs to replicate all the constraints on the state required by the superclass if it manipulates supeclass state directly. This is the reason for favouring private variables and requiring subclasses to use the public interface of the superclass scope or for the superclass to provide a protected interface that does not allow direct manipulation of the state.

Java has the added issue that protected variables are accessible by any class in the same package. This opens up the state of a class very widely and so is generally a huge disadvantage. So in Java, using protected variables is generally a bad thing.

7.6 *What is* **super**?

super is a keyword. It can be used in two ways. As the name of a function called in a subclass constructor, it causes the call of the superclass constructor of the given signature. Used as a variable it is a reference to the superclass cscope of the current object.

7.7 *How does a name get hidden*?

7.8 *What is type conformance*?

7.9 *Why can a reference of type* Object *refer to any kind of object*?

All classes are either direct or indirect subclasses of **class** Object. This means that wherever a reference of type Object is specified an object of any type can be provided, as all objects must be instances of a subclass of Object. The principle that a reference of superclass type can refer to an object of a subclass type follows from this.

7.10 *Why is casting potentially dangerous*?

A cast expression explicitly forces a conversion from one type to another, for example from type Object to type String. At compile time it is typically not possible for the compiler to check that the forced conversion will be valid at runtime. Instead the checking has to be done dynamically when the program is actually run. This means that code that compiled without any type error being reported can still fail with a type error at runtime. Hence, this dangerous in the sense that the programmer cannot rely on the compiler to detect certain kinds of type error and the errors can occur unpredictably when a program is run.

Prior to generics being added to the Java language this meant that retrieving a value from a container like an Arraylist required a cast expression to recover the type of the object reference returned. For example:

```
ArrayList list = new ArrayList ( ) ;
list.add ( "Hello" ) ;
String s = (String) list.get ( 0 ) ;
```

The reference returned by the get method is of type Object as an ArrayList works by storing references of type Object. Without the cast expression the attempted initialization of a String reference with a reference of type Object would result in a compilation error. Even with the cast expression in place it is up to the prgrammer to ensure that the reference returned by get will be of type String.

Generics in Java 5 remove the container problem by allowing the type of the reference values in the container to be specified and then used for compile time type checking:

```
ArrayList<String> list = new ArrayList<String> ( ) ;
list.add ( "Hello" ) ;
String s = list.get ( 0 ) ;
```

Behind the scenes the compiler is actually inserting a cast expression as the generic `ArrayList` still works by storing references of type `Object`. However, the compiler can now guarantee that the cast expression will never result in a runtime type error.

7.11 *How is a method overridden?*

The glib answer is that if a subclass defines a method of the same signature as a method in the superclass then that class is overridden. However, for various reasons, Java actually has quite complicated rules for deciding when a method is overridden. Java has the concept of an override-equivalent method.

7.12 *What are the differences between an abstract class and an interface?*

An interface is limited to declaring abstract instance methods (with no method body), static variables and classes, interfaces and enums. Anything declared in an interface is public. The primary role of an interface is declaring a type that an implementing classes must conform to, allowing mechanisms such as *Programing to an Interface* to be used.

An abstract class must have at least one abstract method, with no method body, but otherwise can declare anything that an non-abstract class can including method bodies. The primary role of an abstract class is to provide a partial implementation that can be inherited and completed by concrete subclasses. This allows common methods and variable to be declared in a single class and not be duplicated across several classes. Like interfaces, abstract classes also declare a type that subclasses must conform to, so can also be used in the same way as an interface in that respect.

Like a class, an interface can extend or inherit from another interface but is not required to. In contrast to the way that all classes are direct or indirect subclasses of **class** `Object`, interfaces are not constrained by a rigid single-inheritance hierarchy. In addition, while a class can inherit from one superclass only, a class can implement any number of interfaces.

7.13 *Describe how shallow and deep copying work.*

The terms shallow and deep copy are only really meaningful for containers. A container must have storage to hold the references to the objects 'contained' in the container. A shallow copy means using different storage for the references to the data but sharing the actual references to the data. Deep copy means having new space to hold the references and creating copies of the data as well. This can mean a lot of copyinf as the data might itslef be a container and so the

notion of deep copy can go very deep indeed. Effectively shallow copy means different storage of references to the shared data and deep copy means no sharing of any objects.

7.14 *What is a nested class?*

A nested class is declared within another class and, hence, is within the scope of the enclosing class. There are several kinds of nested classes, including inner or member classes:

- Top-level nested classes, declared as static. These are ordinary classes that are declared within the scope of another class. They have access to static members of the class they are declared in only.
- Inner or member classes, declared without using static. An inner class has full access to the scope of the class it is declared in and must be created by an object of the inner class.
- Local classes, declared within a local scope.
- Anonymous classes, also declared within a local scope but not given a name.

7.15 *How does a package work?*

Programming Exercises

7.1 *Extend the* `Stack<T>` *class (page 196) to include a method equals that implements value equality for this type. Write a test for the extended class.*

7.2 *Implement matrix multiplication for all the matrix implementation classes shown in Section 7.10, page 255. The multiplication method should not depend upon how a specific kind of matrix is implemented.*

7.3 *Write a program to manage the inventory of a simple warehouse. Class* `Item` *defines the basic properties of an item stored in the warehouse, with subclasses representing real kinds of items. All items have a common set of properties such as size, weight, sell-by date and so on. Allow items to be added to and removed from the warehouse, and also make it possible to display a complete list of the current contents of the warehouse.*

Challenges

7.1 *Starting with class* GraphPanel *shown in* Section 6.8.3, *page* 207, *create a* Graphing *interface declaring a common set of methods to be implemented by class* GraphPanel, PieChartPanel *and* BarChartPanel, *each of which draws a graph or chart of the appropriate form.*

7.2 *By analogy from* Ellipse2D *and* Rectangle2D, *create a class* Triangle2D *for drawing triangles.*