Java Quick Reference Card
CS6501 – Internet Programming – Unit – I – Part - II

Prepared by: B.Bhuvaneswaran / AP (SG) / CSE / REC

Inheritance

- A class that is inherited is called a superclass
- A class that does the inheriting is called a subclass.

The keyword extends indicates that you are making a new class that derives from an existing class. The existing class is called the superclass, base class, or parent class. The new class is called the subclass, derived class, or child class.

General Format

```
class subclass-name extends superclass-name {
        // body of class
}
```

Member Access and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as private.

A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

A Superclass Variable Can Reference a Subclass Object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

super

super has two general forms:

- The first calls the superclass' constructor.
- The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Using super to Call Superclass Constructors

```
super(parameter-list);
```

A Second Use for super

```
super.member
```

Creating a Multilevel Hierarchy

For example, given three classes called A, B, and C, C can be a subclass of B, which is a subclass of A.

Interfaces

Using the keyword interface, you can fully abstract a class' interface from its implementation. That is, using interface, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. In practice, this means that you can define interfaces which don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.

Defining an Interface

```
access interface name {
        return-type method-name1(parameter-list);
        return-type method-name2(parameter-list);
        type final-varname1 = value;
        type final-varname2 = value;
        // ...
        return-type method-nameN(parameter-list);
        type final-varnameN = value;
}
```

Implementing Interfaces

```
access class classname [extends superclass]
                [implements interface [,interface...]] {
        // class-body
}
```

Interfaces Can Be Extended

One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Introducing Nested and Inner Classes

It is possible to define a class within another class; such classes are known as nested classes. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B is known to A, but not outside of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.

There are two types of nested classes:

- static and
- non-static

A static nested class is one which has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

The most important type of nested class is the inner class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do. Thus, an inner class is fully within the scope of its enclosing class.

Proxies

You use a proxy to create at runtime new classes that implement a given set of interfaces. Proxies are only necessary when you don't yet know at compile time which interfaces you need to implement.

The proxy class has the following methods:

- All methods required by the specified interfaces; and
- All methods defined in the Object class (toString, equals, and so on).

An invocation handler is an object of any class that implements the InvocationHandler interface. That interface has a single method:

```
Object invoke(Object proxy, Method method, Object[] args)
```

To create a proxy object, you use the newProxyInstance method of the Proxy class. The method has three parameters:

- A class loader. As part of the Java security model, different class loaders for system classes, classes that are downloaded from the Internet, and so on, can be used.
- An array of Class objects, one for each interface to be implemented.
- An invocation handler.

Proxies can be used for many purposes, such as

- Routing method calls to remote servers;
- Associating user interface events with actions in a running program; and
- Tracing method calls for debugging purposes.

Properties of Proxy Classes

You can also obtain that class with the getProxyClass method:

```
Class proxyClass = Proxy.getProxyClass(null, interfaces);
```

```
class A {
        // ...
}

class B extends A {
        // ...
}

class C extends B {
        // ...
}
```

When Constructors Are Called

Constructors are called in order of derivation, from superclass to subclass.

Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

Dynamic Method Dispatch

Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Using Abstract Classes

Abstract method - General form

```
abstract type name(parameter-list);
```

- Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration.
- There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator. Such objects would be useless, because an abstract class is not fully defined.
- Also, you cannot declare abstract constructors, or abstract static methods.
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract.

Using final with Inheritance

Using final to Prevent Overriding

To disallow a method from being overridden, specify final as a modifier at the start of its declaration. Methods declared as final cannot be overridden.

```
class A {
final void meth() {
        System.out.println("This is a final method.");
        }
}

class B extends A {
void meth() { // ERROR! Can't override.
        System.out.println("Illegal!");
        }
}
```

Using final to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with final. Declaring a class as final implicitly declares all of its methods as final, too.

```
final class A {
        // ...
}

// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
        // ...
}
```

As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

The Object Class

There is one special class, Object, defined by Java. All other classes are subclasses of Object. That is, Object is a superclass of all other classes. This means that a reference variable of type Object can refer to an object of any other class.

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object object)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long milliseconds) void wait(long milliseconds, int nanoseconds)	Waits on another thread of execution.

The methods getClass( ), notify( ), notifyAll( ), and wait( ) are declared as final. You may override the others.

Object Cloning

When you make a copy of a variable, the original and the copy are references to the same object. This means a change to either variable also affects the other

The default cloning operation is "shallow" - it doesn't clone objects that are referenced inside other objects.

If the subobject that is shared between the original and the shallow clone is immutable, then the sharing is safe.

Quite frequently, however, subobjects are mutable, and you must redefine the clone method to make a deep copy that clones the subobjects as well.

Reflection

Reflection is the ability of software to analyze itself. This is provided by the java.lang.reflect package and elements in Class. Reflection is an important capability, needed when using components called Java Beans.

Classes Defined in java.lang.reflect

Class	Primary Function
AccessibleObject	Allows you to bypass the default access control checks. (Added by Java 2)
Array	Allows you to dynamically create and manipulate arrays.
Constructor	Provides information about a constructor.
Field	Provides information about a field.
Method	Provides information about a method.
Modifier	Provides information about class and member access modifiers.
Proxy	Supports dynamic proxy classes. (Added by Java 2, v1.3)
ReflectPermission	Allows reflection of private or protected members of a class. (Added by Java 2)