Chapter 1 Declarations, Initialization and Scoping

1.1 Develop code that declares classes (including abstract and all forms of nested classes), interfaces, and enums, and includes the appropriate use of package and import statements (including static imports)

Class declarations define new reference types and describe how they are implemented.

A nested class is any class whose declaration occurs within the body of another class or interface. A top level class is a class that is not a nested class.

A named class may be declared `abstract` and must be declared `abstract` if it is incompletely implemented; such a class cannot be instantiated, but can be extended by subclasses. A class may be declared `final`, in which case it cannot have subclasses. If a class is declared `public`, then it can be referred to from other packages.

Each class except `Object` is an extension of (that is, a subclass of) a single existing class and may implement interfaces.

The body of a class declares members (fields and methods and nested classes and interfaces), instance and static initializers, and constructors. The scope of a member is the entire declaration of the class to which the member belongs. Field, method, member class, member interface, and constructor declarations may include the access modifiers `public`, `protected`, or `private`. The members of a class include both declared and inherited members. Newly declared fields can hide fields declared in a superclass or superinterface. Newly declared class members and interface members can hide class or interface members declared in a superclass or superinterface. Newly declared methods can hide, implement, or override methods declared in a superclass or superinterface.

Field declarations describe class variables, which are incarnated once, and instance variables, which are freshly incarnated for each instance of the class. A field may be declared `final`, in which case it can be assigned to only once. Any field declaration may include an initializer.

Member class declarations describe nested classes that are members of the surrounding class. Member classes may be `static`, in which case they have no access to the instance variables of the surrounding class; or they may be inner classes.

Member interface declarations describe nested interfaces that are members of the surrounding class.

Method declarations describe code that may be invoked by method invocation expressions. A class method is invoked relative to the class type; an instance method is invoked with respect to some particular object that is an instance of the class type. A method whose declaration does not indicate how it is implemented MUST be declared `abstract`. A method may be declared `final`, in which case it cannot be hidden or overridden. A method may be implemented by platform-dependent native code. A `synchronized` method automatically locks an object before executing its body and automatically unlocks the object on return, as if by use of a `synchronized` statement, thus allowing its activities to be synchronized with those of other threads.

Method names may be overloaded.

Instance initializers are blocks of executable code that may be used to help initialize an instance when it is created.

Static initializers are blocks of executable code that may be used to help initialize a class when it is first loaded.

Constructors are similar to methods, but cannot be invoked directly by a method call; they are used to initialize new class instances. Like methods, they may be overloaded.

Class declaration

A class declaration specifies a new named reference type:

```
Class Declaration:
[Modifiers] class Identifier [extends Parent] [implements Interfaces] { }
```

The "Identifier" in a class declaration specifies the name of the class. A compile-time error occurs if a class has the same simple name as any of its enclosing classes or interfaces.

Class modifiers

A class declaration may include class modifiers (one of):

```
public protected private
abstract static final strictfp
```

Not all modifiers are applicable to all kinds of class declarations. The access modifier `public` pertains only to top level classes and to member classes. The access modifiers `protected` and `private` pertain only to member classes within a directly enclosing class declaration. The access modifier `static` pertains only to member classes. A compile-time error occurs if the same modifier appears more than once in a class declaration. If two or more class modifiers appear in a class declaration, then it is customary, though not required, that they appear in the order consistent with that shown above.

Abstract classes

An `abstract` class is a class that is incomplete, or to be considered incomplete. Only abstract classes may have `abstract` methods, that is, methods that are declared but not yet implemented. If a class that is not `abstract` contains an `abstract` method, then a compile-time error occurs. A class `C` has `abstract` methods if any of the following is true:

- `C` explicitly contains a declaration of an `abstract` method.
- Any of `C`'s superclasses declares an `abstract` method that has not been implemented in `C` or any of its superclasses.
- A direct superinterface of `C` declares or inherits a method (which is therefore necessarily `abstract`) and `C` neither declares nor inherits a method that implements it.

In the example:

```
abstract class Point {
        int x = 1, y = 1;
        void move(int dx, int dy) {
                x += dx;
                y += dy;
                alert();
        }
        abstract void alert();
}
abstract class ColoredPoint extends Point {
        int color;
}
```

```
class SimplePoint extends Point {
        void alert() { }
}
```

a class Point is declared that must be declared abstract, because it contains a declaration of an abstract method named alert. The subclass of Point named ColoredPoint inherits the abstract method alert, so it must also be declared abstract. On the other hand, the subclass of Point named SimplePoint provides an implementation of alert(), so it need not be abstract.

A compile-time error occurs if an attempt is made to create an instance of an abstract class using a class instance creation expression.

Thus, continuing the example just shown, the statement:

```
Point p = new Point(); // wrong
```

would result in a compile-time error; the class Point cannot be instantiated because it is abstract. However, a Point variable could correctly be initialized with a reference to any subclass of Point, and the class SimplePoint is not abstract, so the statement:

```
Point p = new SimplePoint(); // correct
```

would be correct.

A subclass of an abstract class that is not itself abstract may be instantiated, resulting in the execution of a constructor for the abstract class and, therefore, the execution of the field initializers for instance variables of that class. Thus, in the example just given, instantiation of a SimplePoint causes the default constructor and field initializers for x and y of Point to be executed. It is a compile-time error to declare an abstract class type such that it is not possible to create a subclass that implements all of its abstract methods. This situation can occur if the class would have as members two abstract methods that have the same method signature but different return types.

As an example, the declarations:

```
interface Colorable {
        void setColor(int color);
}
abstract class Colored implements Colorable {
        abstract int setColor(int color);
}
```

result in a compile-time error: it would be impossible for any subclass of class Colored to provide an implementation of a method named setColor, taking one argument of type int, that can satisfy both abstract method specifications, because the one in interface Colorable requires the same method to return no value, while the one in class Colored requires the same method to return a value of type int.

A class type should be declared abstract only if the intent is that subclasses can be created to complete the implementation. If the intent is simply to prevent instantiation of a class, the proper way to express this is to declare a constructor of no arguments, make it private, never invoke it, and declare no other constructors. A class of this form usually contains class methods and variables. The class Math is an example of a class that cannot be instantiated; its declaration looks like this:

```
public final class Math {
        private Math() { }             // never instantiate this class
        // ... declarations of class variables and methods ...
}
```

Inner Classes and Enclosing Instances

An inner class is a nested class that is not explicitly or implicitly declared `static`. Inner classes may not declare `static` initializers or member interfaces. Inner classes may not declare `static` members, unless they are compile-time constant fields.

To illustrate these rules, consider the example below:

```
class HasStatic{
        static int j = 100;
}
class Outer{
        class Inner extends HasStatic{
                static final x = 3;    // ok - compile-time constant
                static int y = 4;      // compile-time error, an inner class
        }
        static class NestedButNotInner{
                static int z = 5;      // ok, not an inner class
        }
        interface NeverInner{}         // interfaces are never inner
}
```

Inner classes may inherit `static` members that are not compile-time constants even though they may not declare them. Nested classes that are not inner classes may declare `static` members freely, in accordance with the usual rules of the Java programming language. Member interfaces are always implicitly `static` so they are never considered to be inner classes.

A statement or expression occurs in a `static` context if and only if the innermost method, constructor, instance initializer, static initializer, field initializer, or explicit constructor statement enclosing the statement or expression is a `static` method, a `static` initializer, the variable initializer of a `static` variable, or an explicit constructor invocation statement.

Any local variable, formal method parameter or exception handler parameter used but not declared in an inner class must be declared `final`, and must be definitely assigned before the body of the inner class.

Inner classes include local, anonymous and non-static member classes. Here are some examples:

```
class Outer {
        int i = 100;
        static void classMethod() {
                final int l = 200;
                class LocalInStaticContext{
                        int k = i; // compile-time error
                        int m = l; // ok
                }
        }
        void foo() {
                class Local { // a local class
                        int j = i;
                }
        }
}
```

The declaration of class `LocalInStaticContext` occurs in a static context-within the static method `classMethod`. Instance variables of class `Outer` are not available within the body of a `static` method. In particular, instance variables of `Outer` are not available inside the body of `LocalInStaticContext`. However, local variables from the surrounding method may be referred to without error (provided they are marked `final`).

Inner classes whose declarations do not occur in a `static` context may freely refer to the instance variables of their enclosing class. An instance variable is always defined with respect to an instance. In the case of instance variables of an enclosing class, the instance variable must be defined with respect to an enclosing instance of that class. So, for example, the class `Local` above has an enclosing instance of class `Outer`. As a further example:

```
class WithDeepNesting {
        boolean toBe;
        WithDeepNesting(boolean b) { toBe = b;}
        class Nested {
                boolean theQuestion;
                class DeeplyNested {
                        DeeplyNested(){
                                theQuestion = toBe || !toBe;
                        }
                }
        }
}
```

Here, every instance of `WithDeepNesting.Nested.DeeplyNested` has an enclosing instance of class `WithDeepNesting.Nested` (its immediately enclosing instance) and an enclosing instance of class `WithDeepNesting` (its 2nd lexically enclosing instance).

Typesafe enumerations

The J2SE 5.0 `enum` declaration looks as follows:

```
public enum MainMenu {FILE, EDIT, FORMAT, VIEW};

enum MachineState { BUSY, IDLE, BLOCKED } // Canonical form

enum Day {
        MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

This approach has many advantages including:

1. It provides strong compile-time type safety
2. It provides a separate namespace for each `enum` type and thus eliminates the need to include a prefix in each constant name
3. Constants are not compiled into clients so you can freely add, remove, or reorder them without recompiling the clients
4. Printed values are informative instead of just numbers
5. Enum constants can be used wherever objects can be used

There are a couple of important things to note about enum declarations. As an example, consider the following declaration:

```
public enum MainMenu {FILE, EDIT, FORMAT, VIEW};
```

1. Keyword `enum` is used to declare an enum type.
2. The word `enum` is reserved and therefore if you have been using it as an identifier, you should adjust your code when compiling with a J2SE 5.0 compiler.

3. The above enum declaration generates a class (MainMenu in the above example), which automatically implements the Comparable<MainMenu> and Serializable interfaces, and provides several members including:
   - Static variables FILE, EDIT, FORMAT, and VIEW.
   - Static method values(), which is an array containing the constants in the enum.
   - Static method valueOf(String) that returns the appropriate enum for the string passed in.
   - Appropriately overloaded equals(), hasCode(), toString(), and compareTo() methods.

Here is a complete example that declares an enumeration and then prints the values:

```
public class Example {
        public enum MainMenu {FILE, EDIT, FORMAT, VIEW}

        public static void main(String[] argv) {
                for (MainMenu menu : MainMenu.values()) {
                        System.out.println(menu);
                }
        }
}
```

And the following segment of code shows another example using the switch statement:

```
for(MainMenu menu : MainMenu.values()) {
        switch(menu) {
                case FILE:
                        System.out.println("FILE Menu");
                        break;
                case EDIT:
                        System.out.println("EDIT Menu");
                        break;
                case FORMAT:
                        System.out.println("FORMAT Menu");
                        break;
                case VIEW:
                        System.out.println("VIEW Menu");
                        break;
        }
}
```

It is worth noting that two classes have been added to java.util in support of enums: EnumSet (a high-performance Set implementation for enums; all members of an enum set must be of the same enum type) and EnumMap (a high-performance Map implementation for use with enum keys).

Properties of the enum type

An enum declaration is a special kind of class declaration:

- It can be declared at the top-level and as static enum declaration.
- It is implicitly static, i.e. no outer object is associated with an enum constant.
- It is implicitly final unless it contains constant-specific class bodies, but it can implement interfaces.
- It cannot be declared abstract unless each abstract method is overridden in the constant-specific class body of every enum constant.

```
// (1) Top level enum declaration
public enum SimpleMeal {
        BREAKFAST, LUNCH, DINNER
}

public class EnumTypeDeclarations {
        // (2) Static enum declaration is OK.
        public enum SimpleMeal {
                BREAKFAST, LUNCH, DINNER
        };

        public void foo() {
                // (3) Local (inner) enum declaration is NOT OK!
                enum SimpleMeal {
                        BREAKFAST, LUNCH, DINNER
                }
        }
}
```

Enum constructors

Each constant declaration can be followed by an argument list that is passed to the constructor of the `enum` type having the matching parameter signature.

An implicit standard constructor is created if no constructors are provided for the `enum` type.

As an `enum` cannot be instantiated using the `new` operator, the constructors cannot be called explicitly.

Example of `enum` constructors:

```
public enum Meal {
        BREAKFAST(7, 30), LUNCH(12, 15), DINNER(19, 45);

        private int hh;

        private int mm;

        Meal(int hh, int mm) {
                assert (hh >= 0 && hh <= 23) : "Illegal hour.";
                assert (mm >= 0 && mm <= 59) : "Illegal mins.";
                this.hh = hh;
                this.mm = mm;
        }

        public int getHour() {
                return hh;
        }

        public int getMins() {
                return mm;
        }
}
```

Methods provided for the enum types

Names of members declared in an `enum` type cannot conflict with automatically generated member names:

- The enum constant names cannot be redeclared.
- The following methods cannot be redeclared:

```
// Returns an array containing the constants of this enum class, in the order
// they are declared.
static <this enum class>[] values()

// Return the enum constant with the specified name
static <this enum class> valueOf(String name)
```

Enum types are based on the java.lang.Enum class which provides the default behavior.

Enums cannot declare methods which override the final methods of the java.lang.Enum class:

- clone(), compareTo(Object), equals(Object), getDeclaringClass(), hashCode(), name(), ordinal().
- The final methods do what their names imply, but the clone() method throws an CloneNotSupportedException, as an enum constant cannot be cloned.

Note that the enum constants must be declared before any other declarations in an enum type.

```
public enum Meal {

        int q = 1; // WRONG ! Compilation error !

        BREAKFAST(7, 30), LUNCH(12, 15), DINNER(19, 45);

        private int hh;
        ...
```

Example of using enum type:

```
public class MealClient {
        public static void main(String[] args) {
                for (Meal meal : Meal.values())
                        System.out.println(meal + " served at " + meal.getHour()
                                + ":" + meal.getMins()
                                + ", has the ordinal value " + meal.ordinal());
        }
}
```

The output will be:

```
BREAKFAST served at 7:30, has the ordinal value 0
LUNCH served at 12:15, has the ordinal value 1
DINNER served at 19:45, has the ordinal value 2
```

Extending enum types: constant-specific class bodies

Constant-specific class bodies define anonymous classes inside an enum type that extend the enclosing enum type.

Instance methods declared in these class bodies are accessible outside the enclosing enum type only if they override accessible methods in the enclosing enum type.

An `enum` type that contains constant-specific class bodies cannot be declared `final`:

```
public enum Meal {
        // Each enum constant defines a constant-specific class body
        BREAKFAST(7, 30) {
                public double mealPrice() {
                        double breakfastPrice = 10.50;
                        return breakfastPrice;
                }
        },
        ...
        // WRONG! Compilation error! Cannot override the final method from Meal
        final double mealPrice() { return 0; }
```

Correct example:

```
public enum Meal {
        // Each enum constant defines a constant-specific class body
        BREAKFAST(7, 30) {
                public double mealPrice() {
                        double breakfastPrice = 10.50;
                        return breakfastPrice;
                }
        },
        ...

        // Abstract method which the constant-specific class body
        abstract double mealPrice();
        ...
```

java.util.EnumSet

Enums can be used in a special purpose `Set` implementation (`EnumSet`) which provides better performance.

All of the elements in an `enum` set MUST come from a single `enum` type.

Enum sets are represented internally as bit vectors.

The `EnumSet` class defines new methods and inherits the ones in the `Set`/`Collection` interfaces.

NOTE, all methods are `static` except for the `clone()` method.

All methods return an `EnumSet<E>`.

The `null` reference CANNOT be stored in an `enum` set.

Method summary for the `abstract class EnumSet<E extends Enum<E>>`:

```
// Creates an enum set containing all of the elements in the specified element
type.
allOf(Class<E> elementType)

// Returns a copy of this set.
clone()
```

```
// Creates an enum set with the same element type as the specified enum set,
// initially containing all the elements of this type that are not contained in
// the specified set.
complementOf(EnumSet<E> s)

// Creates an enum set initialized from the specified collection.
copyOf(Collection<E> c)

// Creates an enum set with the same element type as the specified enum
// set, initially containing the same elements (if any).
copyOf(EnumSet<E> s)

// Creates an empty enum set with the specified element type.
noneOf(Class<E> elementType)

// These factory methods creates an enum set initially containing the specified
// element(s). Note that enums cannot be used in varargs as it is not legal to
// use varargs with parameterized types.
of(E e)
of(E e1, E e2)
of(E e1, E e2, E e3)
of(E e1, E e2, E e3, E e4)
of(E e1, E e2, E e3, E e4, E e5)

// Creates an enum set initially containing all of the elements in the range
// defined by the two specified endpoints.
range(E from, E to)
```

Example of using `java.util.EnumSet`:

```
import java.util.EnumSet;
import static java.lang.System.out;

public class UsingEnumSet {
        enum Day {
                MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
        }

        public static void main(String[] args) {

                EnumSet<Day> allDays = EnumSet.range(Day.MONDAY, Day.SUNDAY);
                out.println("All days: " + allDays);

                EnumSet<Day> weekend = EnumSet.range(Day.SATURDAY, Day.SUNDAY);
                out.println("Weekend: " + weekend);

                EnumSet<Day> oddDays = EnumSet.of(Day.MONDAY, Day.WEDNESDAY,
                                Day.FRIDAY, Day.SUNDAY);
                out.println("Odd days: " + oddDays);

                EnumSet<Day> evenDays = EnumSet.complementOf(oddDays);
                out.println("Even days: " + evenDays);

                EnumSet<Day> weekDays = EnumSet.complementOf(weekend);
                out.println("Week days: " + weekDays);
        }
}
```

Output:

```
All days: [MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY]
Weekend: [SATURDAY, SUNDAY]
Odd days: [MONDAY, WEDNESDAY, FRIDAY, SUNDAY]
Even days: [TUESDAY, THURSDAY, SATURDAY]
Week days: [MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY]
```

java.util.EnumMap

Enums can be used in a special purpose `Map` implementation (`EnumMap`) which provides better performance.

Enum maps are represented internally as arrays.

All of the keys in an `enum` map MUST come from a single `enum` type.

Enum maps are maintained in the natural order of their keys (i.e. the order of the `enum` constant declarations).

The `EnumMap` class re-implements most of the methods in the `Map` interface.

The `null` reference as a key is NOT permitted.

Constructor summary of the class `EnumMap<K extends Enum<K>,V>`:

```
// Creates an empty enum map with the specified key type.
EnumMap(Class<K> keyType)

// Creates an enum map with the same key type as the specified enum map,
// initially containing the same mappings (if any).
EnumMap(EnumMap<K,? extends V> m)

// Creates an enum map initialized from the specified map.
EnumMap(Map<K,? extends V> m)
```

Method summary of the class `EnumMap<K extends Enum<K>,V>`:

```
// Removes all mappings from this map.
void clear()

// Returns a shallow copy of this enum map.
EnumMap<K,V> clone()

// Returns true if this map contains a mapping for the specified key.
boolean containsKey(Object key)

// Returns true if this map maps one or more keys to the specified value.
boolean containsValue(Object value)

// Returns a Set view of the mappings contained in this map.
Set<Map.Entry<K,V>> entrySet()

// Compares the specified object with this map for equality.
boolean equals(Object o)

// Returns the value to which this map maps the specified key, or null
// if this map contains no mapping for the specified key.
V get(Object key)
```

```
// Returns a Set view of the keys contained in this map.
Set<K> keySet()

// Associates the specified value with the specified key in this map.
V put(K key, V value)

// Copies all of the mappings from the specified map to this map.
void putAll(Map<? extends K,? extends V> m)

// Removes the mapping for this key from this map if present.
V remove(Object key)

// Returns the number of key-value mappings in this map.
int size()

// Returns a Collection view of the values contained in this map.
Collection<V> values()
```

Example of using `EnumMap`:

```
import java.util.Collection;
import java.util.EnumMap;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import static java.lang.System.out;

public class UsingEnumMap {

        enum Day {
                MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
        }

        public static void main(String[] args) {
                int[] freqArray = { 12, 34, 56, 23, 5, 13, 78 };

                // Create a Map of frequencies
                Map<Day, Integer> ordinaryMap = new HashMap<Day, Integer>();
                for (Day day : Day.values()) {
                        ordinaryMap.put(day, freqArray[day.ordinal()]);
                }
                out.println("Frequency Map: " + ordinaryMap);
                // Create an EnumMap of frequencies
                EnumMap<Day, Integer> frequencyEnumMap =
                        new EnumMap<Day, Integer>(ordinaryMap);

                // Change some frequencies
                frequencyEnumMap.put(Day.MONDAY, 100);
                frequencyEnumMap.put(Day.FRIDAY, 123);
                out.println("Frequency EnumMap: " + frequencyEnumMap);

                // Values
                Collection<Integer> frequencies = frequencyEnumMap.values();
                out.println("Frequencies: " + frequencies);

                // Keys
                Set<Day> days = frequencyEnumMap.keySet();
                out.println("Days: " + days);
        }
}
```

Output:

```
Frequency Map: {THURSDAY=23, SUNDAY=78, SATURDAY=13, WEDNESDAY=56, FRIDAY=5,
TUESDAY=34, MONDAY=12}
Frequency EnumMap: {MONDAY=100, TUESDAY=34, WEDNESDAY=56, THURSDAY=23,
FRIDAY=123, SATURDAY=13, SUNDAY=78}
Frequencies: [100, 34, 56, 23, 123, 13, 78]
Days: [MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY]
```

Creating and using packages

To make classes easier to find and to use, to avoid naming conflicts, and to control access, programmers bundle groups of related classes and interfaces into packages. A package is a collection of related classes and interfaces providing access protection and namespace management.

The classes and interfaces that are part of the Java platform are members of various packages that bundle classes by function: fundamental classes are in `java.lang`, classes for reading and writing (input and output) are in `java.io`, and so on. You can put your classes and interfaces in packages, too.

To create a package, you put a class or an interface in it. To do this, you put a `package` statement at the top of the source file in which the class or the interface is defined. For example, the following code appears in the source file `Circle.java` and puts the `Circle` class in the `graphics` package:

```
package graphics;

public class Circle extends Graphic implements Draggable {
        ...
}
```

The `Circle` class is a `public` member of the `graphics` package. You must include a `package` statement at the top of every source file that defines a class or an interface that is to be a member of the `graphics` package. So you would also include the statement in `Rectangle.java` and so on:

```
package graphics;

public class Rectangle extends Graphic implements Draggable {
        ...
}
```

The scope of the `package` statement is the entire source file, so all classes and interfaces defined in `Circle.java` and `Rectangle.java` are also members of the `graphics` package. If you put multiple classes in a single source file, only one may be `public`, and it must share the name of the source files base name. Only `public` package members are accessible from outside the package. If you do not use a `package` statement, your class or interface ends up in the default package, which is a package that has no name. Generally speaking, the default package is only for small or temporary applications or when you are just beginning development. Otherwise, classes and interfaces belong in named packages.

Only `public` package members are accessible outside the package in which they are defined. To use a `public` package member from outside its package, you must do one or more of the following:

- Refer to the member by its long (qualified) name.

  You can use a package members simple name if the code you are writing is in the same package as that member or if the members package has been imported. However, if you are trying to use a member from a different package and that package has not been imported, you must use the members qualified name, which includes the package name. This is the qualified name for the `Rectangle` class declared in the `graphics` package:

```
graphics.Rectangle
```

You could use this long name to create an instance of `graphics.Rectangle`:

```
graphics.Rectangle myRect = new graphics.Rectangle();
```

Using long names is okay for one-shot uses. You'd likely get annoyed if you had to write `graphics.Rectangle` again and again. Also, your code would get very messy and difficult to read. In such cases, you can just import the member instead.

- Import the package member

  To import a specific member into the current file, put an `import` statement at the beginning of your file before any class or interface definitions but after the `package` statement, if there is one. Here's how you would import the `Circle` class from the `graphics` package:

```
import graphics.Circle;
```

Now you can refer to the `Circle` class by its simple name:

```
Circle myCircle = new Circle();
```

This approach works well if you use just a few members from the `graphics` package. But if you use many classes and interfaces from a package, you can import the entire package.

- Import the members entire package

  To import all the classes and interfaces contained in a particular package, use the `import` statement with the asterisk (*) wildcard character:

```
import graphics.*;
```

Now you can refer to any class or interface in the `graphics` package by its short name:

```
Circle myCircle = new Circle();
Rectangle myRectangle = new Rectangle();
```

The asterisk in the `import` statement can be used only to specify all the classes within a package, as shown here. It cannot be used to match a subset of the classes in a package. For example, the following does not match all the classes in the graphics package that begin with A:

```
import graphics.A*;      // does not work
```

Instead, it generates a compiler error. With the `import` statement, you can import only a single package member or an entire package. For your convenience, the Java runtime system automatically imports two entire packages: `java.lang` and the current package.

Static imports

The static import feature enables you to import static members from a class or an interface and thus use them without a qualifying name. As an example, consider the following interface that contains two constant values:

```
package com.name;
```

```
interface XYZ {
  public static final double Constant1 = someValue;
  public static final double Constant2 = anotherValue;
}
```

Now, the constants in the `XYZ` interface can be used as follows:

```
public class MyClass implements XYZ {
        ...
        double value = 2 * Constant1;
        ...
}
```

As you can see, a class must implement the interface in order to have access to the constants defined in the interface.

In J2SE 5.0, static import solves this problem as shown in the following example:

```
import static com.name.XYZ.*;

public class MyClass {
        ...
        double value = 2 * Constant1;
        ...
}
```

As another example, consider the following static imports:

```
import static java.lang.Math.*;
import static java.lang.System.*;
```

With these two static imports, you now can use:

```
double r = cos(PI * theta);
```

instead of:

```
double r = Math.cos(PI * theta);
```

and

```
out.println("Hello there");
```

instead of:

```
System.out.println("Hello there");
```

1.2 Develop code that declares an interface. Develop code that implements or extends one or more interfaces. Develop code that declares an abstract class. Develop code that extends an abstract class.

1.3 Develop code that declares, initializes, and uses primitives, arrays, enums, and objects as static, instance, and local variables. Also, use legal identifiers for variable names.

Variable Identifiers

Variable names must be legal Java identifiers:

- Keywords and reserved words may not be used.
- It must start with a letter, dollar sign ($), or underscore (_).
- Subsequent characters may be letters, dollar signs, underscores, or digits.
- It is case sensitive. For example, `mymodel` and `MyModel` are different identifiers.

The following are not legal variable names:

```
My Variable              // Contains a space
9pins                    // Begins with a digit
a+c                      // The plus sign is not an alphanumeric character
testing1-2-3             // The hyphen is not an alphanumeric character
O'Reilly                 // Apostrophe is not an alphanumeric character
OReilly_&_Associates // ampersand is not an alphanumeric character
```

Keywords:

abstract	assert	boolean	break	byte
case	catch	char	class	const
continue	default	do	double	else
enum	extends	false	final	finally
float	for	goto	if	implements
import	instanceof	int	interface	long
native	new	null	package	private
protected	public	return	Short	static
strictfp	super	switch	synchronized	this
throw	throws	transient	True	try
void	volatile	while		

The `goto` keyword is not used.

The `const` is also not used.

The `assert` is a new keyword added with Java 1.4

The `enum` is a new keyword added with Java 5.0

1.4 Develop code that declares both static and non-static methods, and - if appropriate - use method names that adhere to the JavaBeans naming standards. Also develop code that declares and uses a variable-length argument list.

To set or get an instance variable, use methods with the names `setVariable(...)` and `getVariable()` for variables named `variable`:

```
private Object variable;

public void setVariable(Object var) {...}
public Object getVariable() {...}
```

For `boolean` instance variables you may provide a getter method named `is...` or `has...` that returns a `boolean`, that can be used conveniently in `boolean` expressions:

```
private boolean enabled;

public void setEnabled(boolean aBoolean) {...}
public boolean isEnabled() {...}
```

Varargs

In past releases, a method that took an arbitrary number of values required you to create an array and put the values into the array prior to invoking the method.

It is still true that multiple arguments must be passed in an array, but the varargs feature automates and hides the process. Furthermore, it is upward compatible with preexisting APIs. So, for example, a method now has this declaration:

```
method (type a, type b, type ... arguments);
```

The three periods after the final parameter's type indicate that the final argument may be passed as an array or as a sequence of arguments. Varargs can be used only in the final argument position.

See the following program for example:

```
public class VarArgs {

        public static void main(String... args) {
                printUs("ONE", "TWO", "THREE");
                printUs("FOUR", "FIVE");
                printUs(new String[]{"SIX", "SEVEN"});
                printUs(); // empty array is allowed too
        }

        private static void printUs(String... args) {
                System.out.println("Var args method");
                for (String s : args) {
                        System.out.println(s);
                }
        }

        private static void printUs(String arg1, String arg2) {
                System.out.println("Specific two argument method");
                System.out.println(arg1);
                System.out.println(arg2);
        }
}
```

It produces the following output:

```
Var args method
ONE
TWO
THREE
Specific two argument method
FOUR
FIVE
Var args method
SIX
SEVEN
Var args method
```

Purpose: add methods that can be called with variable-length argument list.

Heavily employed in formatting text output, aiding internationalization.

Syntax and semantics:

- the last formal parameter in a method declaration can be declared as:

  `ReferenceType ... FormalParameterName`

- the last formal parameter in the method is then interpreted as having the type:

  `ReferenceType[]`

```
// Method declaration
public static void publish(String str, Object ... data) // Object[]

// Method calls
publish("one");              // ("one", new Object[] {})
publish("one", "two");       // ("one", new Object[] {"two"})
publish("one", "two", 3);    // ("one", new Object[] {"two", new Integer(3)})
```

Some more varargs examples:

```
import static java.lang.System.out;

public class VarargsDemo {

    public static void main(String ... args) {
        int day = 1;
        String month = "February";
        int year = 2005;
        flexiPrint();                  // new Object[] {}
        flexiPrint(day);               // new Object[] {new Integer(day)}
        flexiPrint(day, month);        // new Object[] {new Integer(day), month}
        flexiPrint(day, month, year); // new Object[] {new Integer(day),
                                       // month, new Integer(year)}
    }

    public static void flexiPrint(Object ... data) { // Object[]
```

```
        out.println("No. of elements: " + data.length);
        for (int i = 0; i < data.length; i++) {
            out.print(data[i] + " ");
        }
        out.println();
    }
}
```

The output:

```
No. of elements: 0

No. of elements: 1
1
No. of elements: 2
1 February
No. of elements: 3
1 February 2005
```

Overloading resolution

Resolution of overloaded methods selects the most specific method for execution.

One method is more specific than another method if all actual parameters that can be accepted by the one can be accepted by the other. A method call can lead to an ambiguity between two or more overloaded methods, and is flagged by the compiler. Example:

```
...
// The method 'flipFlop(String, int, Integer)' is ambiguous for the type
// VarargsDemo
flipFlop("(String, Integer, int)", new Integer(4), 2004); // COMPILER ERROR!
...

private static void flipFlop(String str, int i, Integer iRef) {
       out.println(str + " ==> (String, int, Integer)");
}

private static void flipFlop(String str, int i, int j) {
       out.println(str + " ==> (String, int, int)");
}
```

This is a legal example:

```
...
flipFlop("(String, Integer, int)", new Integer(4), 2004); // OK
...

private static void flipFlop(String str, Integer iRef,  int i) {
       out.println(str + " ==> (String, Integer, int)");
}

private static void flipFlop(String str, int i, int j) {
       out.println(str + " ==> (String, int, int)");
}
```

The output will be:

```
(String, Integer, int) ==> (String, Integer, int)
```

Varargs and overloading

The example illustrates how the most specific overloaded method is chosen for a method call:

```
public class VarargsOverloading {

        public void operation(String str) {
                String signature = "(String)";
                out.println(str + " => " + signature);
        }

        public void operation(String str, int m) {
                String signature = "(String, int)";
                out.println(str + " => " + signature);
        }

        public void operation(String str, int m, int n) {
                String signature = "(String, int, int)";
                out.println(str + " => " + signature);
        }

        public void operation(String str, Integer... data) {
                String signature = "(String, Integer[])";
                out.println(str + " => " + signature);
        }

        public void operation(String str, Number... data) {
                String signature = "(String, Number[])";
                out.println(str + " => " + signature);
        }

        public void operation(String str, Object... data) {
                String signature = "(String, Object[])";
                out.println(str + " => " + signature);
        }

        public static void main(String[] args) {
                VarargsOverloading ref = new VarargsOverloading();
                ref.operation("1. (String)");
                ref.operation("2. (String, int)", 10);
                ref.operation("3. (String, Integer)", new Integer(10));
                ref.operation("4. (String, int, byte)", 10, (byte) 20);
                ref.operation("5. (String, int, int)", 10, 20);
                ref.operation("6. (String, int, long)", 10, 20L);
                ref.operation("7. (String, int, int, int)", 10, 20, 30);
                ref.operation("8. (String, int, double)", 10, 20.0);
                ref.operation("9. (String, int, String)", 10, "what?");
                ref.operation("10.(String, boolean)", false);
        }
}
```

The output:

```
1. (String) => (String)
2. (String, int) => (String, int)
3. (String, Integer) => (String, int)
4. (String, int, byte) => (String, int, int)
5. (String, int, int) => (String, int, int)
```

```
6. (String, int, long) => (String, Number[])
7. (String, int, int, int) => (String, Integer[])
8. (String, int, double) => (String, Number[])
9. (String, int, String) => (String, Object[])
10.(String, boolean) => (String, Object[])
```

Varargs and overriding

Overriding of varargs methods does not present any surprises as along as criteria for overriding is satisfied.

```
public class OneSuperclass {
        public int doIt(String str, Integer... data) throws
                java.io.EOFException, java.io.FileNotFoundException { // (1)

                String signature = "(String, Integer[])";
                out.println(str + " => " + signature);
                return 1;
        }

        public void doIt(String str, Number... data) { // (2)
                String signature = "(String, Number[])";
                out.println(str + " => " + signature);
        }
}

import static java.lang.System.out;

public class OneSubclass extends OneSuperclass {

        // public int doIt(String str, Integer[] data)        //  Overridden (a)
        public int doIt(String str, Integer... data) // Overridden (b)
                        throws java.io.FileNotFoundException {
                String signature = "(String, Integer[])";
                out.println("Overridden: " + str + " => " + signature);
                return 0;
        }

        public void doIt(String str, Object... data) { // Overloading
                String signature = "(String, Object[])";
                out.println(str + " => " + signature);
        }

        public static void main(String[] args) throws Exception {
                OneSubclass ref = new OneSubclass();
                ref.doIt("1. (String)");
                ref.doIt("2. (String, int)", 10);
                ref.doIt("3. (String, Integer)", new Integer(10));
                ref.doIt("4. (String, int, byte)", 10, (byte) 20);
                ref.doIt("5. (String, int, int)", 10, 20);
                ref.doIt("6. (String, int, long)", 10, 20L);
                ref.doIt("7. (String, int, int, int)", 10, 20, 30);
                ref.doIt("8. (String, int, double)", 10, 20.0);
                ref.doIt("9. (String, int, String)", 10, "what?");
                ref.doIt("10.(String, boolean)", false);
        }
}
```

The output:

```
Overridden: 1. (String) => (String, Integer[])
Overridden: 2. (String, int) => (String, Integer[])
Overridden: 3. (String, Integer) => (String, Integer[])
4. (String, int, byte) => (String, Number[])
Overridden: 5. (String, int, int) => (String, Integer[])
6. (String, int, long) => (String, Number[])
Overridden: 7. (String, int, int, int) => (String, Integer[])
8. (String, int, double) => (String, Number[])
9. (String, int, String) => (String, Object[])
10.(String, boolean) => (String, Object[])
```

Another method signature:

```
...
public int doIt(String str, Integer[] data)   //  Overridden (a)
// public int doIt(String str, Integer... data) // Overridden (b)
...
```

However, the method will be invoked from superclass:

```
1. (String) => (String, Number[])
2. (String, int) => (String, Number[])
3. (String, Integer) => (String, Number[])
4. (String, int, byte) => (String, Number[])
5. (String, int, int) => (String, Number[])
6. (String, int, long) => (String, Number[])
7. (String, int, int, int) => (String, Number[])
8. (String, int, double) => (String, Number[])
9. (String, int, String) => (String, Object[])
10.(String, boolean) => (String, Object[])
```

1.5 Given a code example, determine if a method is correctly overriding or overloading another method, and identify legal return values (including covariant returns), for the method.

Covariant return types

You cannot have two methods in the same class with signatures that only differ by return type. Until the J2SE 5.0 release, it was also true that a class could not override the return type of the methods it inherits from a superclass. J2SE 5.0 allows covariant return types. What this means is that a method in a subclass may return an object whose type is a subclass of the type returned by the method with the same signature in the superclass. This feature removes the need for excessive type checking and casting.

Let's start with the following class, `ConfusedClass`. The class tries to declare two methods with the same signature. One of the methods returns a `JTextField`, and the other returns a `JPasswordField`.

```
import javax.swing.JTextField;
import javax.swing.JPasswordField;

public class ConfusedClass {

        public JTextField getTextField(){
                return new JTextField();
        }

        public JPasswordField getTextField(){
                return new JPasswordField();
        }
}
```

If you try to compile `ConfusedClass`, you get the following compile error:

```
ConfusedClass.java:10: getTextField() is already defined in ConfusedClass
        public JPasswordField getTextField(){
        ^
        1 error
```

Looking at this situation from the perspective of a class calling `getTextField()`, you can see the reason for the compile time error. How would you indicate which of the two methods you are targeting? Consider, for example, this snippet:

```
ConfusedClass cc = new ConfusedClass();
JTextField field = cc.getTextField();
```

Because a `JPasswordField` extends `JTextField`, either version of the method could correctly be called.

Next, create two classes, each of which having a different version of the `getTextField()` methods. The two methods differ by implementation and return type. Start with the following base:

```
import javax.swing.JTextField;

public class ConfusedSuperClass {
        public JTextField getTextField(){
                System.out.println("Called in " + this.getClass());
                return new JTextField();
        }
}
```

Compile `ConfusedSuperClass`. You'll see that it compiles without error. Now create a derived class, one that extends `ConfusedSuperClass`. The derived class attempts to return an instance of `JPasswordField` instead of the `JTextField` returned by the `getTextField()` method in `ConfusedSuperClass`.

```
import javax.swing.JPasswordField;

public class ConfusedSubClass extends ConfusedSuperClass {
        public JPasswordField getTextField(){
                System.out.println("Called in " + this.getClass());
                return new JPasswordField();
        }
}
```

If you use a version of the JDK prior to J2SE 5.0, `ConfusedSubClass` will not compile. You will see an error like this:

```
ConfusedSubClass.java:5: getTextField() in ConfusedSubClass
cannot override getTextField() in ConfusedSuperClass;
attempting to use incompatible return type
found   : javax.swing.JPasswordField
required: javax.swing.JTextField
        public JPasswordField getTextField(){
        ^
        1 error
```

The error reported is that you are attempting to use an incompatible return type. In fact, the `JPasswordField` you are attempting to return is a subtype of `JTextField`. This same code compiles correctly under J2SE 5.0. **You are now allowed to override the return type of a method with a subtype of the original type.** In the current example, the `getTextField()` method in `ConfusedSuperClass` returns an instance of type `JTextField`. The `getTextField()` method in the `ConfusedSubClass` returns an instance of type `JPasswordField`.

You can exercise these two classes with the following `NotConfusedClient` class. This class creates an instance of type `ConfusedSuperClass` and of type `ConfusedSubClass`. It then calls `getTextField()` on each instance, and displays the type corresponding to the object returned by the method:

```
import javax.swing.JTextField;

public class NotConfusedClient {
        static JTextField jTextField;

        public static void main(String[] args) {
                System.out.println("===== Super Class =====");
                jTextField = new ConfusedSuperClass().getTextField();
                System.out.println("Got back an instance of "
                        + jTextField.getClass());

                System.out.println("===== Sub Class =====");
                jTextField = new ConfusedSubClass().getTextField();
                System.out.println("Got back an instance of  "
                        + jTextField.getClass());
        }
}
```

Compile and run `NotConfusedClient`. When you run it, you should see the following output:

```
===== Super Class =====
Called in class ConfusedSuperClass
Got back an instance of class javax.swing.JTextField
===== Sub Class =====
Called in class ConfusedSubClass
Got back an instance of  class javax.swing.JPasswordField
```

In fact, you will get the same output if you change the return type of `getTextField()` to `JTextField` in `ConfusedSubClass`. The payoff comes when you use the object that is returned by the call to `getTextField()`. Before J2SE 5.0, you needed to downcast to take advantage of methods that are present in the derived class but not in the base class. As you've seen, in J2SE 5.0 `ConfusedSubClass` compiles with a different return type specified for `getTextField()` than is present in the superclass. You can now use your covariant return type to call a method that is only available in the subtype. First, recast the supertype like this:

```
public class SuperClass {
        public SuperClass getAnObject(){
                return this;
        }
}
```

Add the exclusive method to the corresponding subclass, and change the return type of the `getAnObject()` method:

```
public class SubClass extends SuperClass {

        public SubClass getAnObject(){
                return this;
        }

        public void exclusiveMethod(){
                System.out.println("Exclusive in Subclass.");
        }

        public static void main(String[] args) {
                System.out.println("===== Call Exclusive method =====");
                new SubClass().getAnObject().exclusiveMethod();
        }
}
```

Compile `SuperClass` and `SubClass`, then run `SubClass`. You should see the following output:

```
===== Call Exclusive method =====
Exclusive in Subclass.
```

The `main()` method creates an instance of the subclass. It then calls `getAnObject()`. It follows this with a call to `exclusiveMethod()` on the returned object of type `SubClass`. If the return type of `getAnObject()` was `SuperClass` in `SubClass.java`, the code would not have compiled.

Example of correct covariant return types:

```
Object <-- Number <-- Integer

import static java.lang.System.out;

class A {
    Object f() {
        return new Object();
    }
}

class B extends A {
    Number f() {       // OK ! A.f() return (Object) is a superclass of B.f()
                       // return (Number)
        return Double.valueOf(1.0d);
    }
}

class C extends B {
    Integer f() {      // OK ! B.f() return (Number) is a superclass of C.f()
                       // return (Integer)
        return 2;
    }
}

public class Client {
    public static void main(String ... sss) {
        A a = new A();
        A b = new B();
        A c = new C();

        out.println("a.f() instanceof Object : " + (a.f() instanceof Object));
        out.println("a.f() instanceof Number : " + (a.f() instanceof Number));
        out.println("b.f() instanceof Number : " + (b.f() instanceof Number));
        out.println("b.f() instanceof Integer : " + (b.f() instanceof Integer));
        out.println("c.f() instanceof Integer : " + (c.f() instanceof Integer));
    }
}
```

The output:

```
a.f() instanceof Object : true
a.f() instanceof Number : false
b.f() instanceof Number : true
b.f() instanceof Integer : false
c.f() instanceof Integer : true
```

Example of wrong covariant return types:

```
           +-- String
Object <---|
           +-- Numeric <-- Integer

'Integer' is NOT a subclass of 'String'


class A {
    Object f() {
        return new Object();
    }
}
```

```
class B extends A {
    String f() {
        return "sss";
    }
}

class C extends B {
    Integer f() { // WRONG ! Compilation error ! The return type is incompatible
with B.f()
        return 2;
    }
}
```

1.6 Given a set of classes and superclasses, develop constructors for one or more of the classes. Given a class declaration, determine if a default constructor will be created, and if so, determine the behavior of that constructor. Given a nested or non-nested class listing, write code to instantiate the class.

Creating Inner Classes Instances

Non-static inner classes have a hidden reference to the enclosing class instance. This means you must have an instance of the enclosing class to create the inner class. You also have to use a special "`new`" function that correctly initializes the hidden reference to the enclosing class. The special "`new`" function is a member of the enclosing class.

```
public class InnerClassTest {
        public class ReallyInner {
        }
}

InnerClassTest o = new InnerClassTest();
InnerClassTest.ReallyInner i = o.new ReallyInner();
```

or

```
InnerClassTest.ReallyInner i = new InnerClassTest().new ReallyInner();
```

Example:

```
public class InnerClassTest {
        public void foo() {
                System.out.println("Outer class");
        }

        public class ReallyInner {
            public void foo() {
                System.out.println("Inner class");
            }

            public void test() {
                this.foo();
                InnerClassTest.this.foo();
            }
        }

        public static void main(String[] args) {
                InnerClassTest.ReallyInner i =
                               new InnerClassTest().new ReallyInner();
                i.test();
        }
}
```

Output:

```
Inner class
Outer class
```