**Oracle Certified Associate Java SE 8 Programmer 1: Revision Notes**

# Chapter 4: Designing Methods

## 4.1 Designing Methods

### 4.1.1 Method Declarations

* The method declaration specifies all the information needed to call the method
* We can add the final keyword before or after the access modifier
* A typical method signature looks like this:

|  |
| --- |
|  |

* As we can see it begins with an optional access modifier
* Before the method name is always the return type (void if there is none)
* An optional throws statement

### 4.1.2 Access Modifiers

Java has four access modifiers:

1. **Public**: method is accessible anywhere
2. **Private**: method can only be called in same class
3. **Protected**: method can only be called in the same package but also outside the package provided it’s in contained in a subclass. NOTE: importing a package does **not** make the methods accessible
4. **Default**: method is accessible to any class in the same package

### 4.1.3. Optional Specifiers

* Optional specifiers are typically declared after the access modifier but can also be place before the access modifier as is the case with final
* We can declare multiple optional specifiers (e.g., public static final method(){})
* The optional specifiers include:

1. static
2. abstract
3. final

### 4.1.4 Return Type

* The return type declared by the method must be returned
* If the method’s return type is void, you can still use the return keyword (e.g., public void myMethod() {return;}
* You must return an object of the same type, i.e. you can not return a long for a method with return type int

### 4.1.5 Method Name

* The method name can only contain letters, digits, $, \_
* The method name can NOT start with a digit

## 4.2 Working with Varargs

* Your method declaration can accept a variable number of parameters, this is a vararg
* A vararg is declared as the type followed by … followed by parameter name
* Example
  + public void walk1(int... nums) { }
* Varargs must be the final parameter of a method as well as being the only vararg in the method declaration
* public void walk4(int... start, int... nums) { } // DOES NOT COMPILE
* Java internally treats the varargs as an array, so we can also pass array objects into the method call
* Example:
  + walk(1, new int[] {4, 5});
* We can NOT pass a null as a vararg
  + walk(1, null); // throws a NullPointerException
* We can access a vararg element using the square brackets like a normal array

16: public static void run(int... nums) {

17: System.out.println(nums[1]);

18: }

## 4.3 Applying Access Modifiers

### 4.3.1: Private Access

* If a class has private methods and variables, they can not be accessed outside that declared class regardless of the package we are in

**Example:**

|  |
| --- |
| 1: package pond.duck;  2: public class FatherDuck {  3: private String noise = "quack";  4: private void quack() {  5: System.out.println(noise); **// private access is ok**  6: }  7: private void makeNoise() {  8: quack(); // **private access is ok**  9: } } |

* We **can** import and instantiate this class
* We can **NOT** call its methods or access its variables (e.g. noise)

|  |
| --- |
| 1: package pond.duck;  2: public class BadDuckling {  3: public void makeNoise() {  4: FatherDuck duck = new FatherDuck();  5: duck.quack(); **// DOES NOT COMPILE**  6: System.out.println(duck.noise); **// DOES NOT COMPILE**  7: } } |

### 4.3.2 Public Access

* Members are accessible anywhere!

### 4.3.3 Default (Package Private Access)

* Default access means we can access methods and variables provided we’re in the same package

**Example**

|  |
| --- |
| package pond.duck;  public class MotherDuck {  String noise = "quack";  void quack() {  System.out.println(noise);  }  private void makeNoise() {  quack();  } } |

**A class in the package:**

|  |
| --- |
| package pond.duck;  public class GoodDuckling {  public void makeNoise() {  MotherDuck duck = new MotherDuck();  duck.quack(); **// default access**  System.out.println(duck.noise); **// default access**  } } |

**A class outside the package**

|  |
| --- |
| package pond.swan;  import pond.duck.MotherDuck; **// import another package**  public class BadCygnet {  public void makeNoise() {  MotherDuck duck = new MotherDuck();  duck.quack(); **// DOES NOT COMPILE**  System.out.println(duck.noise); **// DOES NOT COMPILE**  } } |

### 4.3.4 Protected Access

* Protected methods and variables can be accessed by classes in the same class and by subclasses

**Example**

|  |
| --- |
| package pond.shore;  public class Bird {  protected String text = "floating";  protected void floatInWater() {  System.out.println(text);  } } |

**A Subclass in a different package:**

|  |
| --- |
| package pond.goose;  import pond.shore.Bird; **// in a different package**  public class Gosling extends Bird { **// subclass of Bird**  public void swim() {  floatInWater(); **// calling protected member**  System.out.println(text); **// calling protected member**  } } |

**We cannot access the Bird Class’s members directly, there must be an object of the sub class:**

|  |
| --- |
| 1: package pond.swan; **// in different package than Bird**  2: import pond.shore.Bird;  3: public class Swan extends Bird { **// but subclass of bird**  4: public void swim() {  5: floatInWater(); **// package access to superclass**  6: System.out.println(text); **// package access to superclass**  7: }  8: public void helpOtherSwanSwim() {  9: Swan other = new Swan();  10: other.floatInWater(); **// package access to superclass**  11: System.out.println(other.text); **// package access to superclass**  12: }  13: public void helpOtherBirdSwim() {  14: Bird other = new Bird();  15: other.floatInWater(); **// DOES NOT COMPILE**  16: System.out.println(other.text); **// DOES NOT COMPILE**  17: }  18: } |

# Chapter 5: Class Design

**Objectives:**

* Describe inheritance and is benefits
* Develop code that demonstrates the use of polymorphism including method overriding and object type versus reference type
* Determine when casting is required
* Use super and this keywords to access objects and call constructors
* Use abstract classes and interfaces

## 5.1 Introducing Class Inheritance

### 5.1.1 Extending a Class

* Extending a class creates another class which inherits all public and protected class member – default access members also if the subclass is in the same package

### 5.1.2 Applying Class Access Modifiers

* A class in a java file can only be given **public OR default access**
* Hence, private, or protected access modifiers do not apply to top level classes (inner classes can allow this but this is out of scope)

### 5.1.3 Creating Java Objects

* All Java classes inherit from **java.lang.Object**
* If you define a class which is not extending another class, the compiler automatically inserts **extends java.lang.Object**

### 5.1.4 Defining Constructors

* If a defined class contains **no defined constructors** then the compiler automatically inserts the **no-argument constructor**
* You can call a constructor in the current class using **this()** or from the super class using **super()**
* **The first line of EVERY constructor is a call to a constructor in same/parent class**
* If no call is defined in first statement of constructor, **compiler will automatically insert** **super()**

#### Understanding Compiler Enhancements

* If your class does not have a no-argument constructor (either defined manually or inserted by compiler) AND you extend the class without specifying a call to a specific constructor, the compiler will automatically insert a default no argument constructor whose first line is super() – this will yield a compiler error
* **Example**:

|  |
| --- |
| public class Mammal {  public Mammal(int age) { }  }  public class Elephant extends Mammal {  // DOES NOT COMPILE  } |

* Therefore we must have an explicitly call to the parent constructor in each child node’s constructor

#### Calling Constructors

* When calling a constructor, the parent constructor is always executed before the child constructor

### 5.1.5 Calling Inherited Class Members

* Public/protected members can be called directly from the subclass
* You can call inherited members using this or super
* You can NOT call non-inherited members (members specific to the subclass) using supe

### 5.1.6 Inheriting Methods

* Inheriting methods means there can be collisions in names of methods specific to the subclass and those from the parent class

#### Overriding Methods

* When a method with the same signature is defined in the subclass, we get **method overriding**
* There are rules to method overriding which are a consequence of Java supporting polymorphism

1. Methods must have same signatures
2. The method in the child class must be at least as accessible
3. The method in the child class cannot throw a new exception or a broader than any exception thrown in parent method
   1. If a method is thrown by parent class then the child class is not obligated to throw an exception
   2. The child class can throw any exception which is a subclass of the exception thrown by the parent class method
4. Return type must be covariant

#### Redeclaring Private Methods

* Since private methods are inaccessible outside the class, they can be redeclared without abiding to the rules of overriding methods

#### Hiding Static Methods

* When we override a static method, we must override it with a static method in the child class. This is known as **method hiding** and follows the same rules as method overriding
* It includes an additional rule:
  + If the parent’s method is static then the child’s method must be static

∴ You cannot mark the child method as static if the parent’s method is not static

#### Overriding Vs Hiding Methods

* With method overriding, any call to the method will use the child’s overridden implementation
* With method hiding, only calls from the child classes is where the method is hidden

#### Creating Final Methods

* Final methods **can not be overridden**

### 5.1.7 Inheriting Variables

* **Java does not allow for variables to be overridden, only hidden**
* Hence, both the parent and child copies of a variable are different

## 5.2 Creating Abstract Classes

### 5.2.1 Defining an Abstract Classes

* Abstract classes **cannot be instantiated** only extended
* They **contain abstract methods** which must be overridden by the first concrete class which extends it
* Implementing abstract methods follow the same rule as method overriding
* By definition, an abstract class **can not be private or final**
* By definition, an abstract method **can not be private or final**
* Can contain non-abstract methods

### 5.2.2 Creating a Concrete Class

* The first non-abstract class must implement all inherited abstract methods

### 5.2.3 Extending an Abstract Class

* An abstract class **can extend an abstract class**
  + It can define some additional abstract methods or override an abstract method
* An abstract class **can not extend a concrete class**

## 5.3 Implementing Interfaces

### 5.3.1 Defining an Interface

* Interfaces **cannot be instantiated**, only extended, or implemented
* They contain **implicitly abstract methods**
* The **methods are implicitly public and abstract**, you can not mark them private or final, additionally you can not mark it protected
* Interfaces **can not be private or final**, only default or public
* An interface can contain default methods which have implementations

### 5.3.2 Inheriting an Interface

* An interface **can be extended to another interface**
* An interface **can not be extended by an abstract class**
* An **abstract class** **can implement an interface**
* The first concrete class which implements an interface must implement all inherited methods
* The first concrete class which extends an abstract class which extends an interface must implement all inherited methods
* An **interface can not implement another interface**

### 5.3.3 Abstract Methods and Multiple Inheritance

* Multiple inheritance opens the door to name collisions, when exactly can a class implement multiple interfaces?
* Answer: most of the time
* Counter-example: if two interfaces contain methods with the same method signature but different return types then you can not implement both interfaces without a compiler error

### 5.3.4 Interface Variables

* Interface variables are by definition **public, static and final**

### 5.3.5 Default Interface Methods

* Default methods **contain an implementation**
* **Not assumed to be static, final, or abstract**
* **Assumed to be public** like regular interface methods

### 5.3.6 Static Interface Methods

* Static interface methods are **not inherited by implementing classes**
* **Assumed public,** cannot be marked private
* Can only be called by referencing the interface

## 5.4 Understanding Polymorphism

### 5.4.1 Polymorphism

* **Java supports polymorphism**, meaning an **object can take on many different forms**
* A Java object can be accessed using a reference with the same type as the object, a reference that is a superclass of the object or if the or a reference which defines an interface the object implements

#### Example 1

|  |
| --- |
| public class A { }  public class B extends A {  public static void main(String[] args) {  A a = new B();  }  } |

#### Example 2

|  |
| --- |
| public interface A { }  public class B implements A {  public static void main(String[] args) {  A a = new B();  }  } |

### 5.4.2 Object vs. Reference

* Since java.lang.Object is the superclass of all classes in Java, we can reference any object using it

|  |
| --- |
| Lemur lemur = new Lemur();  Object lemurAsObject = lemur; |

* We no longer have access to methods which are specific to the class Lemur

#### Rules of Polymorphism

1. **The type of the object determines which properties exist within the object in memory**
2. **The type of reference determines which properties and variables are accessible to the Java program**

### 5.4.3 Casting Objects

* If we want to reference an object using a more specific class, we must do an **explicit cast**

|  |
| --- |
| **Lemur lemur = new Lemur();**  **Primate primate = lemur; // does not require explicit cast as primate is a super class**  **Lemur lemur2 = (Lemur)lemur; // explicit cast needed** |

#### Rules for Casting Objects

1. **Casting to a super class does not require explicit cast – but is allowed**
2. **Casting to sub class does require explicit cast**
3. **Compiler will prevent casts to unrelated types**
4. **Exceptions can still be thrown at runtime, if object being cast is not actually an instance of that class**

|  |
| --- |
| **public class Rodent {}**  **public class Capybara extends Rodent {**  **public static void main(String[] args) {**  **Rodent rodent = new Rodent();**  **Capybara capybara = (Capybara)rodent; }} // throws runtime exception**  **}**  **}** |

### 5.4.4 Polymorphic Parameters

* An application of polymorphism is the ability to define methods using interfaces and super classes as parameters
* If a method takes an interface as a parameter, then any class which implements the interface can be passed as a parameter
* Similarly, you can pass subclasses to a method as a parameter
* This enables code reuse

### 5.4.5 Polymorphism and Method Overriding

* Polymorphism is the reason we have rules for method overriding