Java Class Design



**Access modifiers**

**Public access**

* Memebres of interface are implicitly public. If you decalre them private it does not compile.
* Accesable from anywhere
* top level class interface or enum can only be defined as public or default access.

**Protected access**

The members of the class defined as **protected** are visible to:

* Classes and interfaces in same package
* Classes derived from the class or interface, even if they are in seperate package
* Classes in different package can access members of class they are extending**, but they cannot do it via instance variables**(see example in gupta.protectedexample.building) .

**Default access (package access)**

* Can be accesed by any class in same package even if they are not related via inheritance

**Private access**

* Most restrictive

**General**

* Method parameters or local variables (variable in a method) cannot be defined with access modifier. They are in scope or not.
* If accessibility is decreased, it can break code.



**Overloading**

* To resolve overloaded methods, **the compiler considers types of reference variable used to refer the object** (example in class Employee.java). Calls are resolved during compilation. Aka – **bound at compile time.**

Employee e = new CEO();  
Travel.bookTicket(e);-- method bookTicket from Employee.class is called

* Overloaded methods **must differ in parameter list.** Return variable is not taken into consideration, though may vary. Access modifier (private, protected) as well as nonaccess modifier (final, static) may vary, but you cannot base overloading on this.

**Overloading constructors**

* Call to overloaded constructor must be the first thing in code
* Invoked using keeyword **this. Class name does not work.**
* Cannot be defined just by change in access modifier, but **can have different access modifiers**
* Compiler recognizes recursive constructor invocation

**Instance initializer**

* Uses brackets {}
* Fire before class constructors. See example Demo.java
  + Benefits:
    - Makes sense for big class to initialize variables after declaration
    - Regardless of constructors variables are initialized
* Fired after parent constructor call (Parent initializer, parent constructor, child initializer, child constructor)

**Overridding**

* Static methods cannot be overriden, but can be hidden by using the same name and signature

Book b = new CourseBook(true);

* the compile time uses the reference type for the method check. (Book)
* the runtime uses the instance (object) type for the method invocation. (CourseBook)
* aka **bound at runtime**
  + Virtual method invocation is invocation of the correct method–determined using the object type and not its reference.

**Rules of overriding**

- A derived class can assign the same or more access but not weaker access than the overriding method in the derived class.  
- A derived class can’t override a base class method marked final.

* to override a method, the parameter list of the methods in the base and derived **classes must be *exactly* the same**(see example in class main.java.classdesign.gupta.overloading.Book.java). If not, you end up overloading not overriding.
* Constructors can’t be overridden because a **base class constructor isn’t inherited** by a derived class.
* The **return type of overriden methods must be the same or covariant**
* Cannot override nonaccesable method. Such as private…

**hashCode**

* If two objects are equal according to method equals(Object), then calling

methodhashCode() on each of the two objects must produce the same integer

result.

* It’s not required that if two objects are unequal according to method equals

(java.lang.Object), that calling method hashCode() on each of the two objectsmust produce distinct integer results.

* + Same hashCode = different equals OK
  + Different hashcode = different equals NG
* Another rule of method hashCode() is that when it’s invoked on the same object

more than once during the execution of a Java application**, hashCode() *must* consistently**

**return the same integer**, provided no information used in the equals() comparisons

on the object is modified.Therefore, Java recommends using immutable objects (don’t change) as keys for collection classes that use the hashing algorithm.

**Casting**



public interface Printable {

void print();

}

public class ShoppingItem {

public void description() {

System.out.println("Shopping Item");

}

}

public class Book extends ShoppingItem implements Printable {

public void description() {

System.out.println("Book");

}

public void print() {

System.out.println("Printing book");

}

}

* Objects of subclasses can be implicitly casted to their base classes or the interfaces that they implement.

Book book = new Book();

Printable printable = book; --OK ( to superclass)

printable.print();

ShoppingItem shoppingItem = book; -- OK ( to superclass)

shoppingItem.description();

Printable printable = new Book();

printable.description(); -- NG (description is not in Printable)

Printable printable = new Book();

((Book) printable).description(); -- OK (explicit cast)



**Implicit casting**

**Implicit upcasting** is allowed. You can assign a reference variable of a derived class to a reference variable of a base class or the interface it implements.

Book book = new Book();

ShoppingItem shoppingItem = book; -- OK book is shopping item

Object object = book(); -- OK book is object

**Implicit downcasting** isn’t allowed. You can’t assign reference variables of a base class

to reference variables of its derived classes or to the interfaces that it doesn’t implement.

ShoppingItem shopingItem = new ShoppingItem();

Book book = shoppingItem();

In the absence of explicit casting, you’ll never getClassCast-

Exception—a RuntimeException.

Implicit downcasting compiles ok, but throws ClassCastException—RuntimeException.

ShoppingItem item = new ShoppingItem();

Book book = (Book)item; CCE

Printable printable = (Printable)item; -- CCE, compiles even thou ShoppingItem does not

implement printable

* You can cast a class that to interface it does not implement but will get CastClassException at runtime. Casting final class to interface it does not implement results in compilation error. Watch out – class **String** is final.
* Casting to **null** is ok. No compilation nor runtime errors.
* If you cast an instance to a class outside its inheritance tree,

you’ll get a compiler error. If you cast an instance to a class within its

inheritance tree, but the types don’t match at runtime, the code willthrow a ClassCastException.

**Points to remember for casting**

* An instance can be implicitly casted to its superclasses or interfaces that it

implements.

* An instance of a nonfinal class can be explicitly casted to any interface at compile

time.

* Classes in the same inheritance tree can be casted to each other using explicit

casting at compile time.

* Objects of classes that don’t form part of the same inheritance tree cannot

be casted.

* Casting to an interface is successful at runtime if the class implements the

interface.

* Casting to a derived class type is successful at runtime if the casted object is

actually a type of the derived class to which it’s casted.

**Operator instanceof**

* the operator instanceof returns false if the reference variablebeing compared to is null.
* theinstanceof operator *never* throws a runtime exception; itreturns either true or false.
* If the instanceof operator uses inconvertibletypes, the code won’t compile.

**Package**

* Import statement does not import the whole package tree
* If you don’t include package statement class is in default package. Members of this default package cannot be imported from some named package.

**Static import**

* Syntax is import static ….