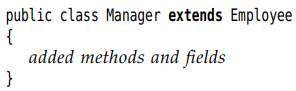
Chapter 5: Inheritance

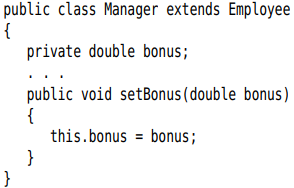
# 5.1 Classes, Superclasses, and Subclasses

## 5.1.1 Defining Subclasses



+ Employee is the superclass, base class, parent class. Manager class is the subclass, derived class, child class.

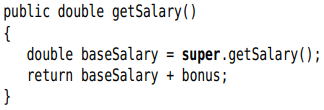
-Subclass has new fields, new methods:



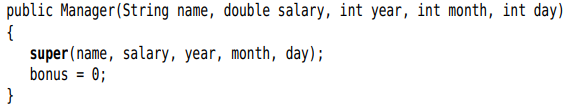
-When defining a subclass, you only need to indicate the *differences* between subclass and superclass.

## 5.1.2 Overriding methods

*-****super***: call the method of superclass



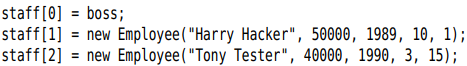
## 5.1.3 Subclass constructors



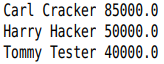
-Example:











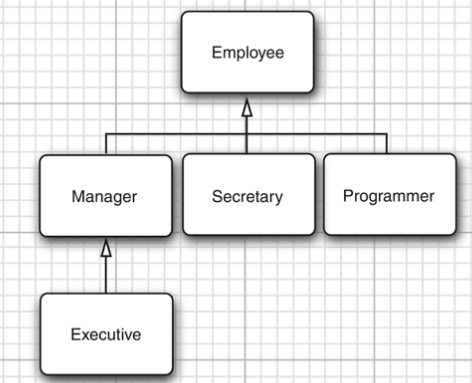
+The virtual machine knows about the actual type of the object to which e refers, and can invoke the correct method.

-The fact that an object variable can refer to multiple actual types is **polymorphism***.* Automatically selecting the appropriate method at runtime is **dynamic binding**

## 5.1.4 Inheritance Hierarchies

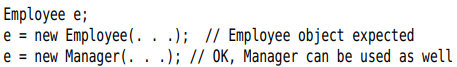
-**Inheritance hierarchies**: The collection of all classes extending a superclass

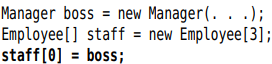
-**Inheritance chain**: path from a class to its ancestors



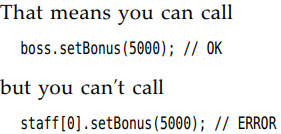
## 5.1.5 Polymorphism

-The “is-a”: every object of subclass is an object of the superclass. **Substituation principle**: you can use a subclass object whenever the program expects a superclass object





+staff[0] and boss refer to same object, but staff[0] is considered to be only an Employee object



-Caution: Arrays of subclass can be converted to arrays of superclass. To make sure no corruption, all arrays remember the element type which they were created

## 5.1.6 Understanding method calls

-x.f(args) (X is an object of class C)

+The compiler looks at the declared type of object and method name. It enumerates all methods called f in the class C and accessible methods called f in superclasses

+The compiler determines the types of arguments in method call. *Overloading resolution*: find the method whose parameters types are best match for arguments

*Method signature*: name and parameter type for method

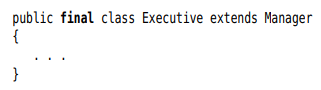
+*static binding*: knows exactly the method private, static, final, constructor.

+**Dynamic binding**. The virtual machine call the version of method for the actual type of object.

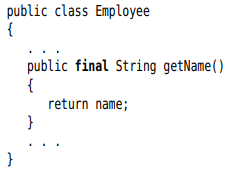
-It’s time-consuming to carry out this search. The virtual machine precomputes for each class a *method table* that lists all method signatures and the actual methods call.

## 5.1.7 Preventing inheritance: final classes and methods

-**final classes**: Classes that **cannot be extended**



-You can also make a **final method** – no subclass can **override** that method.



-Good reason to make a method or class final: its semantics cannot be changed in subclass.

## 5.1.8 Casting

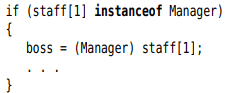
-Casting: the process of forcing a conversion from one type to another



-Convert an object reference from one class to another:



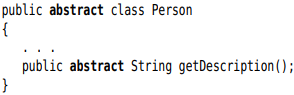
-Good practice: find out whether a cast will succeed before attempting it by **instanceof** operator:

**

-The only reason to make the cast is to use a unique method.

## 5.1.9 Abstract classes

-Moving up the inheritance hierarch, classes become more general and abstract. There are some attributes that make sense for every class. Using the **abstract**, you don’t need to implement the method:



-**Abstract methods** act as placeholders for method that are implemented in subclasses. It can has **instance fields** and **concrete methods**, or **no abstract method**.

-Abstract cannot be instantiated, you can create object variable but refer a nonabstract class. The variable always refers to an object of subclass.



-Abstract methods are an important concept in Java => inside interfaces.

## 5.1.10 Protected access

-**protected:** allow subclass methods to access a superclass field. A protected field is accessible by any class in same package. Use protected fields with caution.

-**Protected methods**: subclasses can be trusted to use

-**Summary**:

+Accessible in the class only (private)

+Accessible by the world (public)

+Accessible in package and subclass (protected)

+Accessible in package – the default

# 5.2 Object: The Cosmic Superclass

Every class in Java extends Object

## 5.2.1 Variables of Type Object

-You can use Object variable to refer objects of any type



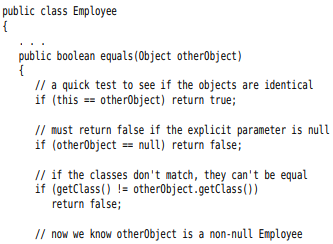
+It just a generic holder for any values. To do sth specific, you need to cast. 

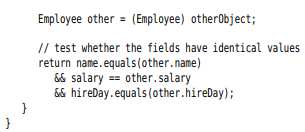
-Primitive types are not Objects

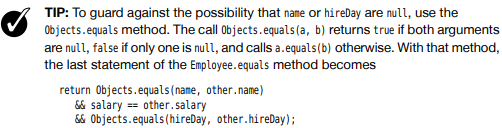
## 5.2.2 The equals method

-**equal** method in Object class test whether one object is considered equal to another. The default determines whether 2 object references are identical.

-We want to implement state-based equality testing:







## 5.2.3 Equality Testing and Inheritance

-Writing the perfect equals method:

equals(Object otherobject)

+Test whether this happens to be identical to otherObject: 

+Test whether otherObject is null: 

+Compare the classes



+Cast otherObject to your class type: 

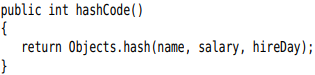
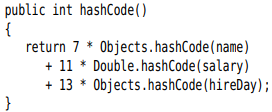
+Compare the fields:



-Note: Use **@Override**. It invoked an error when you override a method with wrong parameters.

## 5.2.4 The hashCode method

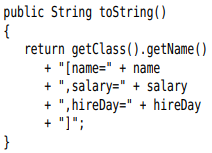
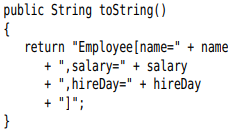
-Hash code is an integer that is derived from an object.



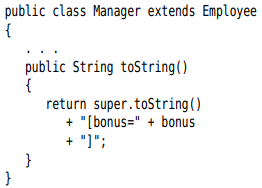
## 5.2.5 The toString method

-toString: return a string representing the value of this object

-Most toString follow this format: the name of class, the field values in []



-Subclass:



-ToString everywhere(system.out.x),string +x

-Caution: with Arrays, use Arrays.toString() or Arrays.deepToString()

-toString method is a great tool for logging.

-We strongly recommend adding a toString method to each class.

# 5.3 Generic Array Lists

-**ArrayList** class is similar to an array, but it automatically adjust its capacity as you add or remove elements. It is a **generic class**with a **type parameter***.*

## 5.3.1 Declaring Array Lists

-Declare and construct an array list:


Or “diamond” syntax:



-**Add** method to add new elements:



+Array list manages an internal array of object references. If you call add method and the internal array is full, the array list creates a bigger array and copies all objects to this array.

-**ensureCapacity** if you know how many elements: 

-Pass an initial capacity to the constructor:



-Note: allocating an array list is not the same as allocating a new array: p250

-**size** method return the current number of elements



-If you are sure that the array list is at its permanent size, call **trimToSize** method. The garbage collector will reclaim any excess memory.

-API note: p251

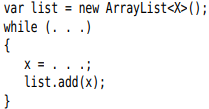
## 5.3.2 Accessing Array List elements

-Set the ith element: 

-Get an element: 

-flexible growth and convenient element access:

+Made an array list and add all the elements



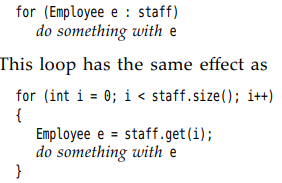
+Use **toArray** to copy elements into array: 

-Add elements in the middle of an array list:

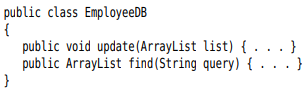


-Remove the element: 

-Traverse the content of array list:



## 5.3.3 Compatibility between Typed and Raw Array Lists



-You can pass a typed array list to update method without any cast:



-You get a warning when you assign a raw ArrayList to a typed one:





-you can tag the variable that receives the cast with the **@SuppressWarnings("unchecked")** annotation.



# 5.4 Object Wrappers and Autoboxing

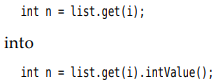
-All primitive types have own **wrappers**: Integer, Long, Float, Double, Short, Byte, Character, and Boolean. (The first six inherit from the common superclass Number.)

-The wrappers are immutable and final.



-*Autoboxing*:  = 

-*Unboxed*:



-Autoboxing and unboxed work with arithmetic expression



-Wrapper class references can throw **NullPointerException**

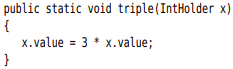


-Mix Integer and Double types:



-Number wrappers put basic method: convert string to number: 

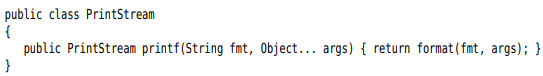
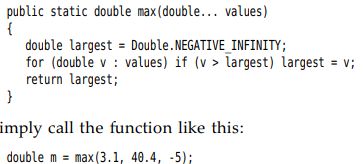
-Note: If you want to change the value of wrapper, use **holder**types org.omg.CORBA:



-API notes: p259

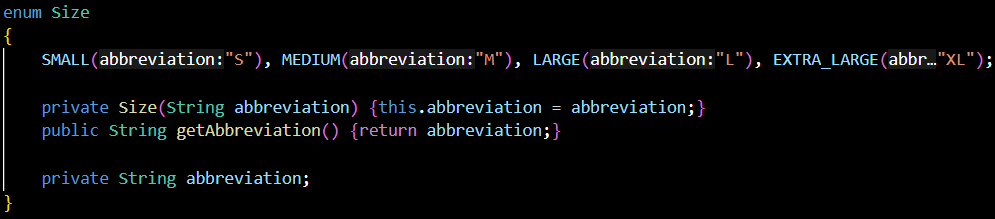
# 5.5 Methods with a Variable number of parameters

-It’s possible to provide methods that can be called with a variable number of parameters. (“varagrs” methods)

# 5.6 Enumeration Classes





+The type Size is a class. It has 4 instances and cannot construct new objects.

# 5.7 Reflection

-A ***reflective*** program can analyze the capabilities of classes. The **reflection library**give a toolset to support UI builders, object-relational mappers to inquire the capabilities of classes.

## 5.7.1 The Class Class

-*runtime type identification*: used by JVM to keep track of the class which each object belongs.

-We can also access this info by Class class.



-Obtain a Class object corresponding to a class name:



-If T is any Java type, T.class is the matching class object



-The JVM manages a unique Class object for each type. You can use == to compare class objects:



-You can use a class object to construct instances of the class:



-API note: p267

## 5.7.2 A Primer on declaring exceptions

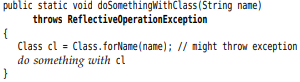
-When an error occurs at runtime, a program can throw an exception. You can provide a *handler* that catches the exception and deals with it.

-There are 2 types of exceptions:

+checked exceptions: the compiler checks that the programmer are aware of exception

+unchecked: the compiler does not expect that a class exists: access a null reference.

-**throw**: used when a method contains a statement that might throw a checked exception



+Any method calling this method needs a throws.

## 5.7.3 Resources

-Classes often have associated data files: image and sound file, text files with message. It’s **resource**

-The **Class** class provides a useful service for locating resource files. p269

## 5.7.4 Using Reflection to analyze the capabilities of classes

-java.lang.reflect: Field, Method, Constructor class

5.7.5 Using reflection to analyze objects at Runtime

5.7.6 Using reflection to write generic array code

5.7.7 Invoking arbitrary methods and constructors

**-Study later**

# 5.8 Design Hints for Inheritance