Inheritance

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

Overriding superclass methods

* public double getSalary()

{

   double baseSalary = **super**.getSalary(); //call superclass method

   return baseSalary + bonus;

}

* Manager boss = new Manager('Carl Cracker', 80000, 1987, 12, 15);

boss.setBonus(5000);

Subclass constructors

* since subclass cannot access private field from superclass.
* calls superclass constructor as **n,s,year,month** as parameter
* public Manager(String name, double salary, int year, int month, int day)

{

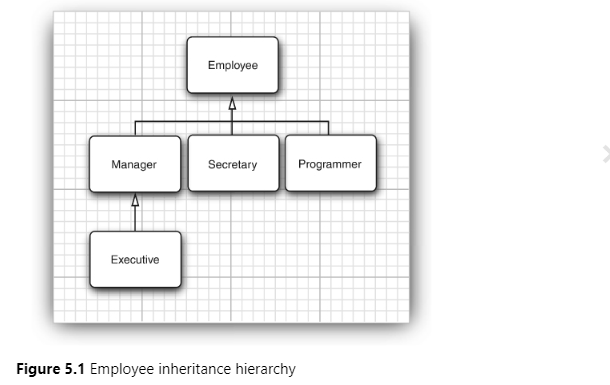
**super**(name, salary, year, month, day);

   bonus = 0;

}

Inheritance hierarchy

* Inheritance need not stop at deriving one layer of classes. We could have an Executive class that extends Manager, for example. The collection of all classes extending a common superclass is called an inheritance hierarchy



Polymorphism

* “is–a” rule states that every object of the subclass is an object of the superclass.
* For example, every manager is an employee. Thus, it makes sense for the Manager class to be a subclass of the Employee class. Naturally, the opposite is not true—not every employee is a manager.
* Employee e = new Employee(. . .);  //Employee object expected

Employee e = new Manager(. . .); //OK, Manager can be used as well

* In the Java programming language, object variables are polymorphic. A variable of type Employee can refer to an object of type Employee or to an object of any subclass of the Employee class (such as Manager, Executive, Secretary and so on)

Preventing Inheritance

* Prevent forming a subclass
* Classes that cannot be extended are called final class
* public **final** class Executive extends Manager

{

   . . .

}

* If you do this, then no subclass can override that method

public class Employee

{

    . . .

    public **final** String getName()

    {

       return name;

    }

    . . .

}

* There is only one good reason to make a method or class final: to make sure its semantics cannot be changed in a subclass. For example, the getTime and setTime methods of the Calendar class are final. This indicates that the designers of the Calendar class have taken over responsibility for the conversion between the Date class and the calendar state. No subclass should be allowed to mess up this arrangement. Similarly, the String class is a final class. That means nobody can define a subclass of String. In other words, if you have a String reference, you know it refers to a String and nothing but a String.

Casting

* var staff = new Employee[3];

staff[0] = new Manager('Carl Cracker', 80000, 1987, 12, 15);

staff[1] = new Employee('Harry Hacker', 50000, 1989, 10, 1);

staff[2] = new Employee('Tony Tester', 40000, 1990, 3, 15);

* Manager boss = (Manager) staff[0];
* The only reason to make the cast is to use a method that is unique to managers, such as setBonus. If for some reason you find yourself wanting to call setBonus on Employee objects, ask yourself whether this is an indication of a design flaw in the superclass. It may make sense to redesign the superclass and add a setBonus method. Remember, it takes only one uncaught ClassCastException to terminate your program. In general, it is best to minimize the use of casts and the instanceof operator.
* To sum up:

You can cast only within an inheritance hierarchy.

Use instanceof to check before casting from a superclass to a subclass.

Instanceof

* if (staff[i] **instanceof** Manager)

{

   Manager boss = (Manager) staff[i];

   boss.setBonus(5000);

}

* Employee e;

if (e instanceof Manager m && m.getBonus() > 10000)

Protected Access

* protected field is accessible by any class in the same package.
* 5.1.10
* In practice, use protected fields with caution. Suppose your class is used by other programmers and you designed it with protected fields.

Variables of type object

* You can use a variable of type Object to refer to objects of any type:

Object obj = new Employee('Harry Hacker', 35000);

The equals method

* The equals method in the Object class tests whether one object is considered equal to another.

Equality Testing and Inheritance, Here is a recipe for writing the perfect equals method:

1. Name the explicit parameter otherObject—later, you will need to cast it to another variable that you should call other.

2. Test whether this happens to be identical to otherObject:

if (this == otherObject) return true;

This statement is just an optimization. In practice, this is a common case. It is much cheaper to check for identity than to compare the fields.

3. Test whether otherObject is null and return false if it is. This test is required.

if (otherObject == null) return false;

4. Compare the classes of this and otherObject. If the semantics of equals can change in subclasses, use the getClass test:

if (getClass() != otherObject.getClass()) return false;

ClassName other = (ClassName) otherObject;

If the same semantics holds for *all* subclasses, you can use an instanceof test:

if (!(otherObject instanceof ClassName other)) return false;

Note that the instanceof test sets other to otherObject if it succeeds. No cast is necessary.

5. Now compare the fields, as required by your notion of equality. Use == for primitive type fields, Objects.equals for object fields. Return true if all fields match, false otherwise.

return field1 == other.field1

   && Objects.equals(field2, other.field2)

   && . . .;

If you redefine equals in a subclass, include a call to super.equals(other).

The hashCode Method

* A hash code is an integer that is derived from an object. Hash codes should be scrambled—if x and y are two distinct objects, there should be a high probability that x.hashCode() and y.hashCode() are different.
* hashCode in Java helps the program to run faster. For example, comparing two objects by their hashcodes will give the result 20 times faster than comparing them using the equals() function.

The toString Method

* returns a string representing the value of this object.

Object wrappers and outboxing

* Occasionally, you need to convert a primitive type like int to an object. All primitive types have class counterparts. For example, a class Integer corresponds to the primitive type int. These kinds of classes are usually called wrappers.
* The wrapper classes are immutable—you cannot change a wrapped value after the wrapper has been constructed. They are also final, so you cannot subclass them.

Abstract classes

* a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
* Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

Enumeration classes

Sealed classes

Reflection

* executing Java program to examine or "introspect" upon itself, and manipulate internal properties of the program. For example, it's possible for a Java class to obtain the names of all its members and display them.

**DESIGN HINTS FOR INHERITANCE**

1. *Place common operations and fields in the superclass.*

This is why we put the name field into the Person class instead of replicating it in the Employee and Student classes.

2. *Don’t use protected fields.*

Some programmers think it is a good idea to define most instance fields as protected, “just in case,” so that subclasses can access these fields if they need to. However, the protected mechanism doesn’t give much protection, for two reasons. First, the set of subclasses is unbounded—anyone can form a subclass of your classes and then write code that directly accesses protected instance fields, thereby breaking encapsulation. And second, in Java, all classes in the same package have access to protected fields, whether or not they are subclasses.

However, protected methods can be useful to indicate methods that are not ready for general use and should be redefined in subclasses.

3. *Use inheritance to model the “is–a” relationship.*

Inheritance is a handy code-saver, but sometimes people overuse it. For example, suppose we need a Contractor class. Contractors have names and hire dates, but they do not have salaries. Instead, they are paid by the hour, and they do not stay around long enough to get a raise. There is the temptation to form a subclass Contractor from Employee and add an hourlyWage field.

public class Contractor extends Employee

{

   private double hourlyWage;

   . . .

}

This is *not* a good idea, however, because now each contractor object has both a salary and hourly wage field. It will cause you no end of grief when you implement methods for printing paychecks or tax forms. You will end up writing more code than you would have written by not inheriting in the first place.

The contractor-employee relationship fails the “is–a” test. A contractor is not a special case of an employee.

4. *Don’t use inheritance unless all inherited methods make sense.*

Suppose we want to write a Holiday class. Surely every holiday is a day, and days can be expressed as instances of the GregorianCalendar class, so we can use inheritance.

class Holiday extends GregorianCalendar { . . . }

Unfortunately, the set of holidays is not *closed* under the inherited operations. One of the public methods of GregorianCalendar is add. And add can turn holidays into nonholidays:

Holiday christmas;

christmas.add(Calendar.DAY\_OF\_MONTH, 12);

Therefore, inheritance is not appropriate in this example.

Note that this problem does not arise if you extend an immutable class. Suppose you have an immutable date class, similar to LocalDate but not final. If you form a Holiday subclass, there is no method that can turn a holiday into a nonholiday.

5. *Don’t change the expected behavior when you override a method.*

The substitution principle applies not just to syntax but, more importantly, to behavior. When you override a method, you should not unreasonably change its behavior. The compiler can’t help you—it cannot check whether your redefinitions make sense. For example, you can “fix” the issue of the add method in the Holiday class by redefining add, perhaps to do nothing, or to throw an exception, or to move on to the next holiday.

However, such a fix violates the substitution principle. The sequence of statements

int d1 = x.get(Calendar.DAY\_OF\_MONTH);

x.add(Calendar.DAY\_OF\_MONTH, 1);

int d2 = x.get(Calendar.DAY\_OF\_MONTH);

System.out.println(d2 - d1);

should have the *expected behavior*, no matter whether x is of type GregorianCalendar or Holiday.

Of course, therein lies the rub. Reasonable and unreasonable people can argue at length about what the expected behavior is. For example, some authors argue that the substitution principle requires Manager.equals to ignore the bonus field because Employee.equals ignores it. These discussions are pointless if they occur in a vacuum. Ultimately, what matters is that you do not circumvent the intent of the original design when you override methods in subclasses.

6. *Use polymorphism, not type information.*

Whenever you find code of the form

if (x *is of type 1*)

    $$$*action*1(x);

else if (x *is of type 2*)

    $$$*action*2(x);

think polymorphism.

Do *action*1 and *action*2 represent a common concept? If so, make the concept a method of a common superclass or interface of both types. Then, you can simply call

x.*action*();

and have the dynamic dispatch mechanism inherent in polymorphism launch the correct action.

Code that uses polymorphic methods or interface implementations is much easier to maintain and extend than code using multiple type tests.

7. *Don’t overuse reflection.*

The reflection mechanism lets you write programs with amazing generality, by detecting fields and methods at runtime. This capability can be extremely useful for systems programming, but it is usually not appropriate in applications. Reflection is fragile—with it, the compiler cannot help you find programming errors. Any errors are found at runtime and result in exceptions.