西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

Object Oriented Programming

Chapter 5
Inheritance

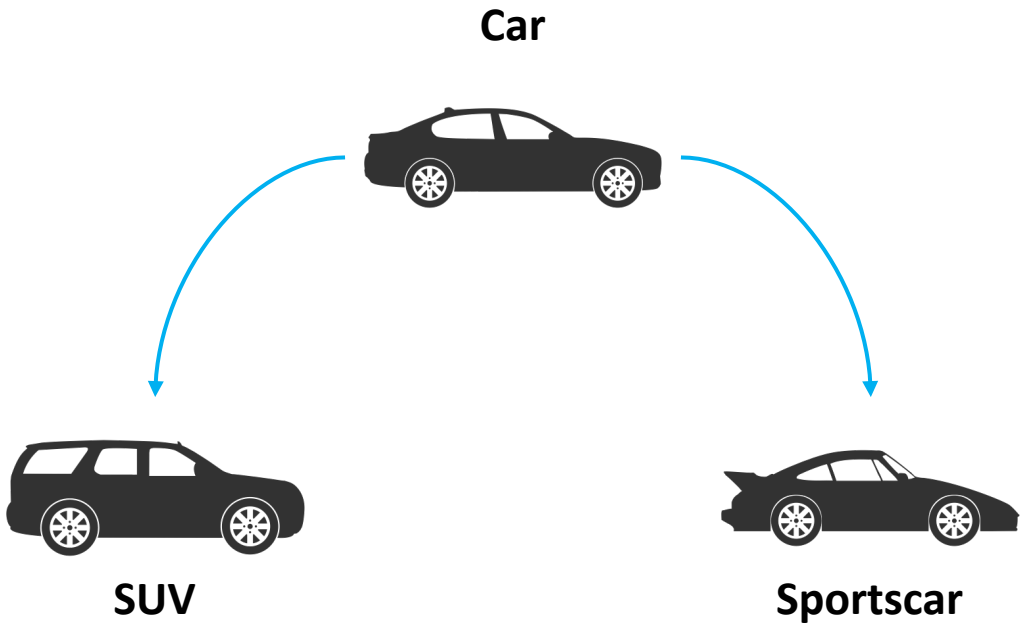
Dr. Helei Cui

18 Apr 2024

Slides partially adapted from lecture notes by Cay Horstmann

Recap

- **Inheritance allows classes to derive from other classes.**

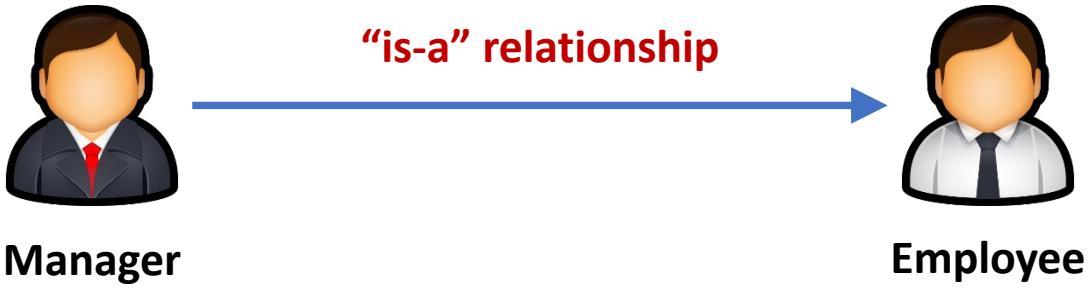


With inheritance, reusability is a major advantage. You can reuse the fields and methods of the existing class.

Contents

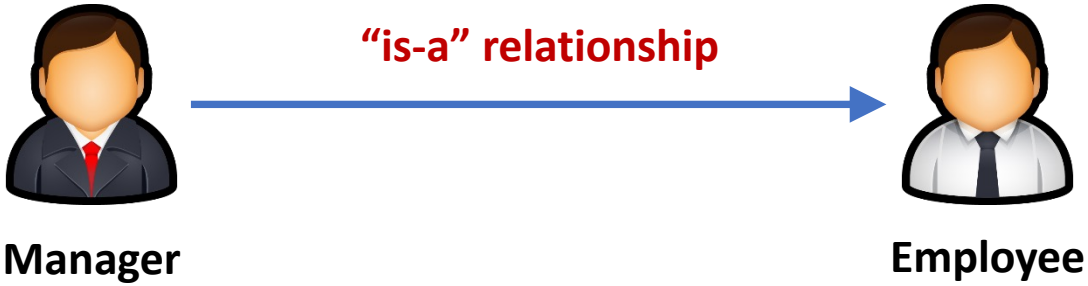
Manager is an Employee

- Background story:
  - In a company, managers are just like employees in many respects. Both employees and managers are paid a salary.
  - However, while employees are expected to complete their assigned tasks in return for receiving their salary, managers get bonuses if they actually achieve what they are supposed to do.



The inheritance, or "is–a" relationship, expresses a relationship between a more special and a more general class.

UML Diagram


"is-a" relationship
Manager
Employee


Manager
-bonus: double
+getSalary(): double
+setBonus(b: double): void
Employee
-name: String
-salary: double
-hireDay: LocalDate
+getName(): String
+getSalary(): double
+getHireDay(): LocalDate
+raiseSalary(byPercent: double): void

5.1.1 Defining Subclasses

- **Use the extends keyword:**

```
public class Manager extends Employee {
    // added methods and fields
}
```

  - The existing class (*i.e., Employee*) is called the ***superclass***, *base class*, or *parent class*.
  - The new class (*i.e., Manager*) is called the ***subclass***, *derived class*, or *child class*.

The prefixes super and sub come from the language of sets used in theoretical computer science and mathematics.

5.1.1 Defining Subclasses

- **Subclasses have more functionality than their superclasses.**
- `Manager` adds a new field to store the bonus, and a new method to set it.

```
public class Manager extends Employee {
    private double bonus;
    public void setBonus(double bonus) {
        this.bonus = bonus;
   }
}
```

- `Manager` inherits the fields and methods from `Employee`:
  - Fields: *name, salary, hireDay,* and *bonus*.
  - Methods: *getName, getHireday, getSalary, raiseSalary,* and *setBonus*.

5.1.1 Defining Subclasses

- When defining a subclass by extending its superclass, you only need to indicate the **differences** between the subclass and the superclass.
  - **General methods ---> superclass**
  - **Specialized methods ---> subclass**
  - *Factoring out common functionality by moving it to a superclass is routine in OOP.*

5.1.2 Overriding Methods

- When an inherited method is not appropriate, you need to override it in the subclass.
- First attempt:

```
public class Manager extends Employee {
    public double getSalary() {
        return salary + bonus; // won't work
    }
}
```

- Reason:
  - Subclass methods cannot access private superclass fields.

5.1.2 Overriding Methods

- Second attempt:

```
public double getSalary() {
    return getSalary() + bonus; // still won't work
}
```

- **Solution: use `super` to avoid recursive call.**

```
public double getSalary() {
    return super.getSalary() + bonus;
}
```

  - "`super`" is not a reference to an object. Instead, `super` is a special keyword that directs the compiler to invoke the superclass method.

A subclass can add fields, and it can add methods or override the methods of the superclass. However, inheritance can never take away any fields or methods.

5.1.3 Subclass Constructors

- **A subclass constructor can invoke a superclass constructor:**

```
public Manager(String name, double salary, int year, int month, int day) {
   super(name, salary, year, month, day);
   bonus = 0;
}
```

  - The call using `super` must be the first statement in the constructor for the subclass.
  - If no explicit call to superclass constructor, no-argument constructor of superclass is invoked.
  - If the superclass does not have a no-argument constructor, the compiler reports an error.

this & super

- The `this` keyword:
  - to denote a reference to the implicit parameter
  - to call another constructor of the same class
- The `super` keyword:
  - to invoke a superclass method
  - to invoke a superclass constructor
- When used to invoke constructors, the this and super keywords are closely related.
  - The constructor calls can only occur as the first statement in another constructor.
  - The constructor parameters are either passed to another constructor of the same class (`this`) or a constructor of the superclass (`super`).

Dynamic Binding in ManagerTest.java

```
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
boss.setBonus(5000);

var staff = new Employee[3]; // make an array of three employees
staff[0] = boss;
staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);

for (Employee e : staff) {
    System.out.println(e.getName() + " " + e.getSalary());
}
```

- The virtual machine knows about the actual type of the object to which e refers, and therefore can invoke the correct method.
  - This is called **(dynamic) polymorphism**, i.e., an object variable (such as the variable e) can refer to multiple actual types.
  - This is a perfect example of dynamic binding as in overriding both parent and child classes have same method and in this case the type of the object determines which method is to be executed. And the type of object is determined at the run time.

5.1.4 Inheritance Hierarchies

- The collection of all classes extending a common superclass is called an *inheritance hierarchy*.

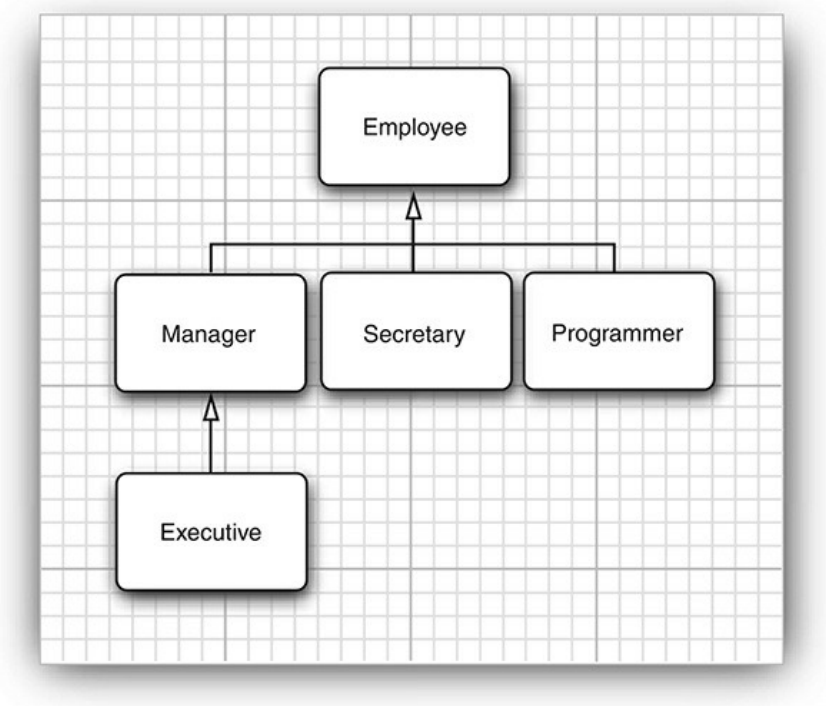


Figure 5.1 Employee inheritance hierarchy

- The path from a particular class to its ancestors in the inheritance hierarchy is its *inheritance chain*.

5.1.5 Polymorphism

- **Simple rule ("is-a" rule):**
  - Every object of the subclass is an object of the superclass.
  - But the opposite is not true.
- **Substitution principle:**
  - You can use a subclass object whenever the program expects a superclass object.
- For example:
  - You can assign a subclass object to a superclass variable.

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

5.1.5 Polymorphism

- **Polymorphism: A variable can refer to multiple types.**
  - A variable of type `Employee` can refer to an object of type `Employee` or to an object of any subclass of the `Employee` class (e.g., `Manager`, `Executive`, and `Secretary`).

```
Manager boss = new Manager(. . .);
Employee[] staff = new Employee[3];
staff[0] = boss;  // staff[0] and boss refer to the same object.
```

  - The variables `staff[0]` and `boss` refer to the same object. **However, `staff[0]` is considered to be only an `Employee` object by the compiler.**

```
boss.setBonus(5000);      // OK
staff[0].setBonus(5000); // ERROR
Manager m = staff[i];     // ERROR
```

  - **You cannot assign a superclass reference to a subclass variable.**
    - Reason: Not all employees are managers.

5.1.6 Understanding Method Calls

- `x.f(args)` - suppose `x` is declared to be of type `C`.
  1. The compiler finds all accessible methods called `f` in `C` and its superclasses.
  2. The compiler selects the method whose parameter types match the argument types (**overloading resolution**).
  3. If the method is `private`, `static`, `final`, or a constructor, then the compiler knows exactly which method to call (**static binding**).
  4. Otherwise, the exact method is found at runtime (**dynamic binding**).

5.1.6 Understanding Method Calls

- The `Employee` method table shows that all methods are defined in the Employee class itself:

```
Employee:
    getName() -> Employee.getName()
    getSalary() -> Employee.getSalary()
    getHireDay() -> Employee.getHireDay()
    raiseSalary(double) -> Employee.raiseSalary(double)
```

- The `manager` method table is slightly different.
  - Three methods are inherited, one method is redefined, and one method is added in Manager table.

```
Manager:
    getName() -> Employee.getName()
    getSalary() -> Manager.getSalary()
    getHireDay() -> Employee.getHireDay()
    raiseSalary(double) -> Employee.raiseSalary(double)
    setBonus(double) -> Manager.setBonus(double)
```

5.1.6 Understanding Method Calls

- At runtime, the call `e.getSalary()` is resolved as follows:
  1. First, the virtual machine fetches the method table for the actual type of `e`. That may be the table for `Employee`, `Manager`, or another subclass of `Employee`.
  2. Then, the virtual machine looks up the defining class for the `getSalary()` signature. Now it knows which method to call.
  3. Finally, the virtual machine calls the method.
- **Dynamic binding makes programs extensible without the need for modifying existing code.**
  - Suppose a new class `Executive` is added and there is the possibility that the variable `e` refers to an object of that class.
  - The code containing the call `e.getSalary()` need not be recompiled.
  - The `Executive.getSalary()` method is called automatically if `e` happens to refer to an object of type `Executive`.

5.1.7 Preventing Inheritance: Final Classes and Methods

- **Final classes cannot be extended.**
  - Use `final` modifier when defining a class.

```
public final class Executive extends Manager {
   . . .
}
```

- **Final methods cannot be overridden by subclasses.**
  - Use `final` modifier.

```
public class Employee {
    . . .
    public final String getName() {
        return name;
    }
}
```

  - All methods (not fields) in a `final` class are automatically `final`.

Use only for design (to make sure its semantics cannot be changed in a subclass) - not necessary for performance.

5.1.8 Casting

- The process of forcing a conversion from one type to another is called casting.

```
double x = 3.405;
int nx = (int) x; // converts the value of x into an integer
```

- Use a similar syntax to make a cast of an object reference.

```
Manager boss = (Manager) staff[0];
boss.setBonus(...); // call Manager methods
```

  - Make a cast is to use an object in its full capacity after its actual type has been temporarily forgotten.
  - You can cast only within an inheritance hierarchy.

5.1.8 Casting

- If `staff[1]` wasn't actually a `Manager`, a `ClassCastException` occur.

```
Manager boss = (Manager) staff[1]; // ERROR
```

- Can test with `instanceof` operator:

```
if (staff[1] instanceof Manager) {
   boss = (Manager) staff[1];
   . . .
}
```

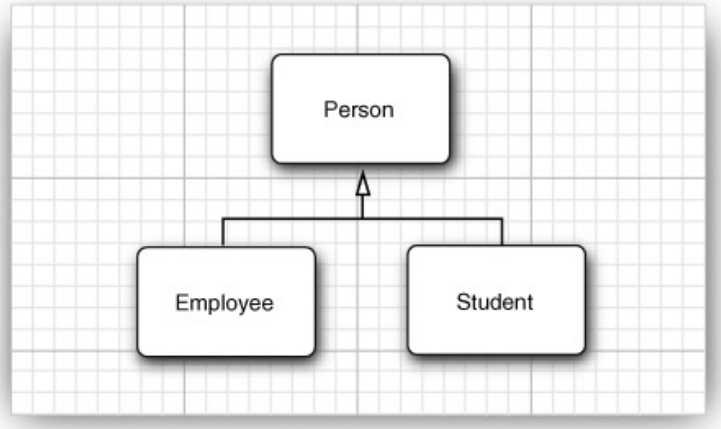
- Compiler won't let you make absurd casts:

```
String c = (String) staff[1]; // Compile-time error
```

The only reason to make the cast is to use a method that is unique to managers, such as *`setBonus`*.

5.1.9 Abstract Classes

- When factoring out common classes, it can become difficult to implement methods in the most general classes.
  - E.g., classes `Employee` and `Student` can have a common superclass `Person`



- Each class defines a `getDescription` method, returning a description string:

```
an employee with a salary of $50,000.00
a student majoring in computer science
```

It is easy to implement this method for the Employee and Student classes. But what information can you provide in the Person class?

5.1.9 Abstract Classes

- Declare method as abstract and don't provide implementation:

```
public abstract String getDescription();
```

- Class with abstract methods must be declared abstract:

```
public abstract class Person
```

- Abstract classes can have fields, constructors, and concrete methods:

```
public abstract class Person {
    private String name;
    public Person(String name) {
        this.name = name;
    }
    public abstract String getDescription();
    public String getName() {
        return name;
    }
}
```

5.1.9 Abstract Classes

- When you extend an abstract class, you have two choices:
  - You can leave some or all of the abstract methods undefined; then you must tag the subclass as abstract as well.
  - Or you can define all methods, and the subclass is no longer abstract.

- **Abstract classes cannot be instantiated!**

```
Person p1 = new Person("Vince Vu");                 // Error!
Person p2 = new Student("Vince Vu", "Economics"); // Ok
```

  - You can still create object variables of an abstract class, but such a variable must refer to an object of a nonabstract subclass.

5.1.10 Protected Access

- Recall that a subclass cannot access the `private` fields of its superclass.
- But a `protected` field or method is accessible from subclasses:

```
public class Employee {
   protected double salary; // Manager methods can access it
}
```

- **Caution**:
  - Protected features have package visibility.
  - Protected fields restrict evolution - anyone can extend a class.
- Protected methods are sometimes useful for methods that are “tricky” to use.
  - This indicates that the subclasses (which, presumably, know their ancestor well) can be trusted to use the method correctly, but other classes cannot.

Access Control Modifiers

1. Accessible in the class only (`private`).
2. Accessible by the world (`public`).
3. Accessible in the package and all subclasses (`protected`).
4. Accessible in the package - the (unfortunate) default. No modifiers are needed.

Contents

5.2.1 Variables of Type Object

- The Object is the superclass of all Java classes.

```
Object obj = new Employee("Harry Hacker", 35000);
Employee e = (Employee) obj;
```

- Only primitive types (numbers, characters, and boolean values) are not objects.
- Arrays (both objects and primitive types) are class types that extend the Object class.

```
Employee[] staff = new Employee[10];
obj = staff;                          // OK
obj = new int[10];                    // OK
```

5.2.2 The equals Method

- **The `equals` method tests whether two object references are identical.**
  - You can **override** it to test if two employees equal if they have the same name, salary, and hire date.

```
public class Employee {
    public boolean equals(Object otherObject) {
        // a quick test to see if the objects are identical
        if (this == otherObject) return true;
        // must return false if the explicit parameter is null
        if (otherObject == null) return false;
        // if the classes don't match, they can't be equal
        if (getClass() != otherObject.getClass()) return false;
        // now we know otherObject is a non-null Employee
        Employee other = (Employee) otherObject;
        // test whether the fields have identical values
        return name.equals(other.name)
            && salary == other.salary
            && hireDay.equals(other.hireDay);
    }
}
```

5.2.2 The equals Method

- Static `Objects.equals` method is null safe.
  - Returns true if both arguments are null, false if only one is null.

```
return Objects.equals(name, other.name)
   && salary == other.salary
   && Object.equals(hireDay, other.hireDay);
```

- When you define the equals method for a subclass, first call equals on the superclass.
  - If that test doesn't pass, then the objects can't be equal.

```
public class Manager extends Employee {
   . . .
   public boolean equals(Object otherObject) {
      if (!super.equals(otherObject)) return false;

      Manager other = (Manager) otherObject;
      return bonus == other.bonus;
   }
}
```

5.2.3 Equality Testing and Inheritance

- The equals method needs to be:
  - *Reflexive* - `x.equals(x)` returns true.
  - *Symmetric* - if `x.equals(y)`, then `y.equals(x)`.
  - *Transitive* - if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`.
  - *Consistent* - If the objects to which `x` and `y` refer haven't changed, then `x.equals(y)` always return the same value.
  - For any non-null reference `x`, `x.equals(null)` return `false`.

5.2.3 Equality Testing and Inheritance

- Hard to do with mixed types.
  - If e is an Employee and m is a Manager, then m.equals(e) and e.equals(m) must be the same.
  - If the meaning of equals must be fixed in the superclass:

```
public class ClassName {
    public final boolean equals(Object otherObject) {

        if (this == otherObject) return true;

        if (otherObject == null) return false;

        if (!(otherObject instanceOf ClassName)) return false;

        ClassName other = (ClassName) otherObject;

        return id == other.id;
    }
}
```

5.2.4 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.

String	Hash Code
Hello	69609650
Harry	69496448
Hacker	-2141031506

Table 5.1 Hash Codes Resulting from the hashCode Method

- Hash code computation in the `String` class:

```
int hash = 0;
for (int i = 0; i < length(); i++) {
    hash = 31 * hash + charAt(i);
}
```

5.2.4 The hashCode Method

- **Every object has a default hash code. Object.hashCode is derived from memory address.**

```
var s = "Ok";
var sb = new StringBuilder(s);
System.out.println(s.hashCode() + " " + sb.hashCode());
var t = new String("Ok");
var tb = new StringBuilder(t);
System.out.println(t.hashCode() + " " + tb.hashCode());
```

Object	Hash Code	Object	Hash Code
s	2556	t	2556
sb	20526976	tb	20527144

- Override `hashCode()` whenever you override `equals()`.
- Combine the hash codes of the fields that the `equals` method compares.
  - If `x.equals(y)` is true, then `x.hashCode()` must return the same value as `y.hashCode()`.

```
public class Employee {
    . . .
    public int hashCode() {
        return Objects.hash(name, salary, hireDay);
    }
}
```

5.2.5 The toString Method

- **Return a string representing the value of this object.**
  - *E.g., "java.awt.Point[x=10,y=20]" is the output of the Point class's toString method.*
  - You can call `getClass().getName()` to obtain a string with the class name.

```
public String toString()
{
    return getClass().getName()+ "[name=" + name + ",salary=" +
    salary + ",hireDay=" + hireDay + "]";
}
```

- The subclass should define its own `toString` method and add the subclass fields.

```
public class Manager extends Employee {
    . . .
    public String toString() {
        return super.toString() + "[bonus=" + bonus + "]";
    }
}
```

5.2.5 The `toString` Method

- When you concatenate a string and an object, the `toString()` method is automatically invoked on the object.

```
var p = new Point(10, 20);
String message = "The current position is " + p;
// automatically invokes p.toString()
```

  - The Object class defines the `toString()` method to print the class name and the hash code of the object.

```
System.out.println(System.out);
// outputs java.io.PrintStream@2f6684
```

We strongly recommend that you add a `toString()` method to each class that you write.

Contents

Generic Array Lists

- In C /C++, you have to fix the sizes of arrays at compile time.
- In Java, you can set the size of an array at runtime.

```
int actualSize = . . .;
var staff = new Employee[actualSize];
```

- The length of an array is fixed - inconvenient when it is unknown in advance.
  - **Solution in Java: `ArrayList`**
  - `ArrayList` class manages an Object[] array that grows and shrinks on demand.
  - It is a ***generic class*** with a type parameter:
    - Use a type parameter such as `ArrayList<Employee>` to specify element type.

5.3.1 Declaring Array Lists

- Can omit type parameter in the constructor:

```
var staff = new ArrayList<Employee>();    // use the var keyword
ArrayList<Employee> staff = new ArrayList<>(); // diamond syntax
```

- Use `add` method to add object to the end:

```
staff.add(new Employee("Harry Hacker", . . .));
```

- The call `staff.size()` yields the current size.

```
staff.ensureCapacity(100); // capacity is 100
new ArrayList<>(100)       // capacity is 100
new Employee[100]          // size is 100

staff.size()               // for array list
a.length                   // for array
```

- Call the `trimToSize` method to make the array list at its permanent size.
  - You should only use this when you are sure you won't add any more elements to the array list.

5.3.2 Accessing Array List Elements

- Access and modify elements with the `get` and `set` methods:

```
Employee e = staff.get(i);
staff.set(i, tony);          //equivalent to a[i] = harry
```

- Can use "for-each" loop to visit elements:

```
for (Employee e : staff) {
    System.out.println(e);
}
```

- **To get flexible growth and convenient element access:**

```
var list = new ArrayList<X>();
while (. . .) {
    x = . . .;
    list.add(x);
}
var a = new X[list.size()];   // define an array
list.toArray(a);              // copy into array
```

5.3.3 Compatibility between Typed and Raw Array Lists

- Suppose you have the following legacy class:

```
public class EmployeeDB {
    public void update(ArrayList list) { . . . }
    public ArrayList find(String query) { . . . }
}
```

- You can pass a typed array list to the `update` method without any casts.

```
ArrayList<Employee> staff = . . .;
employeeDB.update(staff);          // no problem
```

- Conversely, when you assign a raw `ArrayList` to a typed one, you get a warning.

```
ArrayList<Employee> result = employeeDB.find(query); // warning
```

5.3.3 Compatibility between Typed and Raw Array Lists

- Using a cast does not make the warning go away.

```
ArrayList<Employee> result = (ArrayList<Employee>)
employeeDB.find(query);                    // yields another warning
```

  - For compatibility, the compiler translates all typed array lists into raw ArrayList objects after checking that the type rules were not violated.
  - In a running program, all array lists are the same - there are no type parameters in the virtual machine.
  - Thus, the casts (ArrayList) and (ArrayList<Employee>) carry out identical runtime checks.

- Tag the variable that receives the cast with the `@SuppressWarnings("unchecked")` annotation.

```
@SuppressWarnings("unchecked")
ArrayList<Employee> result = (ArrayList<Employee>)
employeeDB.find(query);                    // yields another warning
```

Contents

Object Wrappers and Autoboxing

- `ArrayList` can only hold objects, not `int` values.
- An object of the `Integer` wrapper class wraps an `int` value.
- Conversion between `int` and `Integer` is automatic:

```
var list = new ArrayList<Integer>();
list.add(3);          // same as list.add(Integer.valueOf(3))
int n = list.get(i); // same as int n = list.get(i).intValue()
```

  - **The conversion is called *autoboxing*.**
- The wrapper classes have obvious names: `Integer`, `Long`, `Float`, `Double`, `Short`, `Byte`, `Character`, and `Boolean`.
  - The first six inherit from the common superclass `Number`.
  - 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.

Object Wrappers and Autoboxing

- Conversely, when you assign an `Integer` object to an `int` value, it is automatically **unboxed**.
  - The compiler translates "*`int n = list.get(i);`*" into "*`int n = list.get(i).intValue();`*".
- Automatic boxing and unboxing even works with arithmetic expressions.

```
Integer n = 3;
n++;
```

- `==` (only tests whether the objects have identical memory locations) doesn't work with wrappers.

```
Integer a = 1000;
Integer b = 1000;
if (a == b) . . . // probably fail
```

  - **Call the `equals` method to compare wrapper objects.**

Object Wrappers and Autoboxing

- Since wrappers can be `null`, it is possible for autounboxing to throw a `NullPointerException`:

```
Integer n = null;
System.out.println(2 * n); // throws NullPointerException
```

- If you mix `Integer` and `Double` types, the `Integer` value is unboxed, and boxed into a `Double`:

```
Integer n = 1;
Double x = 2.0;
System.out.println(true ? n : x); // prints 1.0
```

- To convert a string to an integer:

```
int x = Integer.parseInt(s); //parseInt is a static method
```

- The wrapper classes can't be used to implement methods that modify numeric parameters.
  - Because parameters to Java methods are always **passed by value**.
  - Even using `Integer` still fails, because `Integer` objects are **immutable**.

Contents

Methods with a Variable Number of Parameters

- Some methods take a variable number of arguments.

```
System.out.printf("%d", n);
System.out.printf("%d %s", n, "widgets");
```

- Variable arguments are indicated with ellipsis.

```
public class PrintStream {
    . . .
    public PrintStream printf(String fmt, Object... args)
    { . . . }
}
```

  - The method actually receives two parameters:
    - the format string;
    - an `Object[]` array that holds all other parameters.

```
System.out.printf("%d %s", new Object[] { new Integer(n),
"widgets" } );
```

Methods with a Variable Number of Parameters

- You can define your own "variable arguments" methods.

```
public static double max(double... values) {
    double largest = Double.NEGATIVE_INFINITY;
    for (double v : values) if (v > largest) largest = v;
    return largest;
}
```

- Call the method like this:

```
double m = max(3.1, 40.4, -5);
```

  - The compiler passes a "`new double[] { 3.1, 40.4, -5 }`" to the `max` function.

Contents

Enumeration Classes

- Enumeration class defines all instances.

```
public enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
```

  - This is a class, with four instances.
  - Use == to compare them (no need to use `equals`).

- Also, you can add constructors, methods, and fields:

```
public enum Size {
    SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");
    private String abbreviation;
    private Size(String abbreviation) {
        this.abbreviation = abbreviation;
    }
    public String getAbbreviation() {
        return abbreviation;
    }
}
```

- **The constructor of an enumeration is always private.**

Enumeration Classes

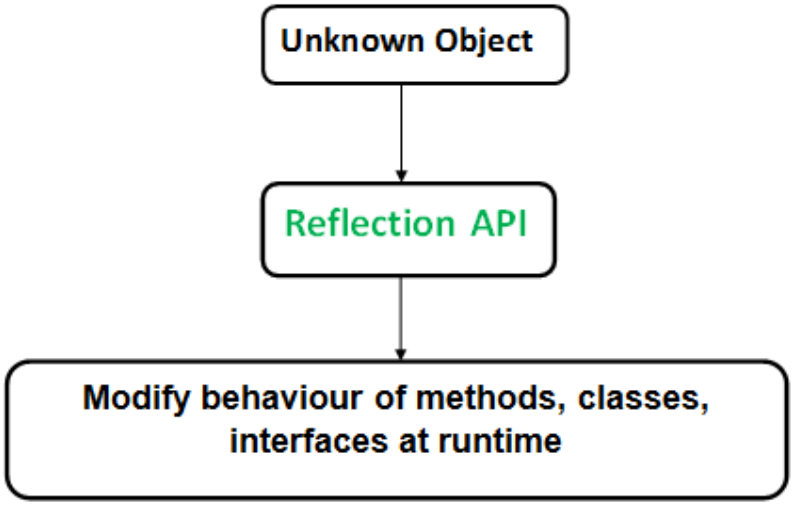
- All enumeration classes are subclasses of `Enum` and inherit methods:
  - `toString()` - yields the name "SMALL", "MEDIUM", ...
  - `ordinal()` - yields the position 0, 1, ...

```
String toString()
// returns the name of this enumerated constant.
int ordinal()
// returns the zero-based position of this enumerated constant
in the enum declaration.
```

Contents

Reflection

- Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime.
  - The required classes for reflection are provided under java.lang.reflect package.
  - Reflection gives us information about the class to which an object belongs and also the methods of that class which can be executed by using the object.
  - Through reflection we can invoke methods at runtime irrespective of the access specifier used with them.



https://www.geeksforgeeks.org/reflection-in-java/

Reflection

- Reflection can be used to get information about
  - **Class** - The getClass() method is used to get the name of the class to which an object belongs.
  - **Constructors** - The getConstructors() method is used to get the public constructors of the class to which an object belongs.
  - **Methods** - The getMethods() method is used to get the public methods of the class to which an objects belongs.

- Example: https://www.geeksforgeeks.org/reflection-in-java/

Reflection

- **Advantages of Using Reflection:**
  - **Extensibility Features:** An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.
  - **Debugging and testing tools:** Debuggers use the property of reflection to examine private members on classes.
- **Drawbacks:**
  - **Performance Overhead:** Reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.
  - **Exposure of Internals:** Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.

https://www.geeksforgeeks.org/reflection-in-java/

Contents

Design Hints for Inheritance

1. **Place common operations and fields in the superclass.**
   - Recall the `name` field in the `Person` class, so you are not required to replicate it in the `Employee` and `Manager` classes.

2. **Don't use `protected` fields.**
   - Can be accessed by subclasses.
   - Could be dangerous:
     - 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.
     - In Java, all classes in the same package have access to `protected` fields.
   - **Can be useful to indicate methods that are not ready for general use and should be redefined in subclasses.**

Design Hints for Inheritance

3. **Use inheritance to model the "is-a" relationship.**
   - Do the "is-a" test.

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 { . . . }
. . .
Holiday christmas;
christmas.add(Calendar.DAY_OF_MONTH, 12);
```

   - 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.

Design Hints for Inheritance

5. **Don't change the expected behavior when you override a method.**
   - *Can you "fix" the issue of the `Holiday` **add** method by redefining it?*
     - **Answer is NO!**
   - The following code should work no matter whether `x` is a `GregorianCalendar` or `Holiday` object:

```
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);
```

6. **Use polymorphism, not type information.**

```
if (x is of type 1)
   action1(x);
else if (x is of type 2)
   action2(x);
```

```
x.action();
```

   - *Do action1 and action2 represent a common concept?* If so, use polymorphism.

Recap

- 5.1 Classes, Superclasses, and Subclasses
- 5.2 Object: The Cosmic Superclass
- 5.3 Generic Array Lists
- 5.4 Object Wrappers and Autoboxing
- 5.5 Methods with a Variable Number of Parameters
- 5.6 Enumeration Classes
- 5.7 Reflection (Optional)
- 5.8 Design Hints for Inheritance