Lecture 11-2

Class Inheritance

Inheritance

Background

- *Inheritance* is a major feature of object-oriented programming
- We'll give an introduction to inheritance, focusing on main concepts and techniques.

Two forms of inheritance

Class inheritance (this lecture)

- Interface inheritance (next lecture)

Inheritance

Requirement (example): Develop a system that manages *instructors*, *students*, and *courses*.

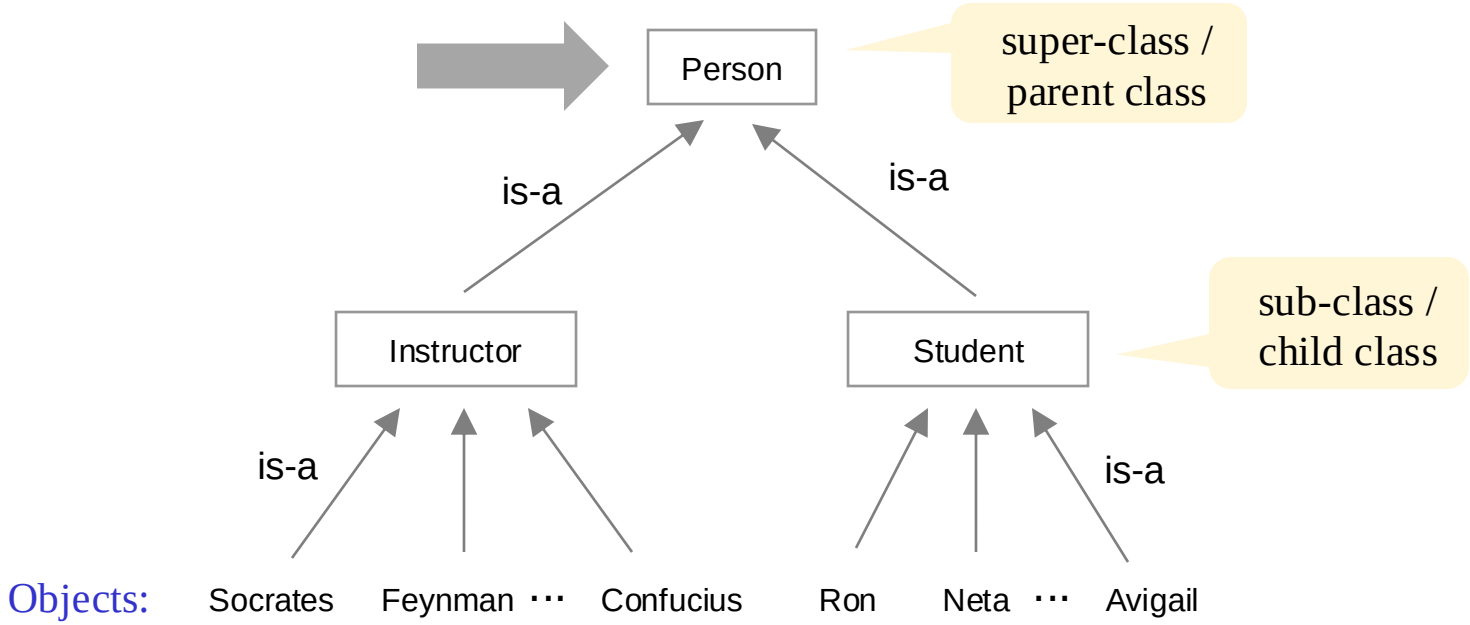
Observation

Instructors and *students* have common properties: *name*, *address*, *birth date*, ...

and common operations: *getAddress*, *setAddress*, ...

Insight: Create...

- a "super-class" that represents the common properties (*fields*) and operations (*methods*), and
- "sub-classes" that add *instructor*-specific and *student*-specific properties and operations.



Super-class: Person

```
/** Represents a person. */
public class Person {
   // Person fields
 protected String name;
   protected String address;
    /** Constructs a Person */
 public Person(String name, String address) {
      this.name = name;
      this.address = address;
   }
    /** Sets the address of this person. */
 public void setAddress(String address) {
      this.address = address;
   }
   /** Textual description of this person. */
 public String toString() {
      name + ", " + address;
   }
   // More Person methods...
}
```

Represents the common properties and operations that every person in the university has.

Member (field / method) visibility:

`private`: visible only in this class

`protected`: also visible by sub-classes

`public`: visible by any class

No modifier: package-private (later)

Sub-class: Instructor

super-class

```
/** Represents a person. */
public class Person {

   // Person fields
 protected String name;
   protected String address;

   /** Constructs a Person */
 public Person(String name, String address) {
      this.name = name;
      this.address = address;
   }

   /** Sets the address of this person. */
 public void setAddress(String address) {
      this.address = address;
   }

   /** Textual description of this person. */
 public String toString() {
      name + ", " + address;
   }
   // More Person methods...
}
```

sub-class

```
/** Represents an instructor. */
public class Instructor extends Person {
   // Instructor-specific fields:
   private String title;

   /** Constructs an instructor. */
   public Instructor(String name, String address,
                     String title) {
      super(name, address)
      this.title = title;

   }

   /** Textual description of this instructor. */
   @Override
   public String toString() {
      return title + " " + super.toString();
   }

   // More Instructor methods...
}
```

`extends` Person

Inherits all the Person members (fields + methods)

Plus, adds instructor-specific properties and operations.

Sub-class: Instructor

super-class

```
/** Represents a person. */
public class Person {

   // Person fields
 protected String name;
   protected String address;

   /** Constructs a Person */
 public Person(String name, String address) {
      this.name = name;
      this.address = address;
   }

   /** Sets the address of this person. */
 public void setAddress(String address) {
      this.address = address;
   }

   /** Textual description of this person. */
 public String toString() {
      name + ", " + address;
   }
   // More Person methods...
}
```

sub-class

```
/** Represents an instructor. */
public class Instructor extends Person {
   // Instructor-specific fields:
   private String title;

   /** Constructs an instructor. */
   public Instructor(String name, String address,
                     String title) {
      super(name, address)
      this.title = title;
   }

   /** Textual description of this instructor. */
   @Override
   public String toString() {
      return title + " " + super.toString();
   }

   // More Instructor methods...
}
```

Typical sub-class constructor:

Starts by calling the super-class constructor to create the object and handle the super-class fields. Then handles the sub-class fields.

Unlike other class members, constructors are not inherited;

A sub-class constructor must begin by calling a super constructor.

Sub-class: Instructor

super-class

```
/** Represents a person. */
public class Person {

   // Person fields
 protected String name;
   protected String address;

   /** Constructs a Person */
 public Person(String name, String address) {
      this.name = name;
      this.address = address;
   }

   /** Sets the address of this person. */
 public void setAddress(String address) {
      this.address = address;
   }

   /** Textual description of this person. */
 public String toString() {
      name + ", " + address;
   }
   ...
}
```

sub-class

```
/** Represents an instructor. */
public class Instructor extends Person {
   // Instructor-specific fields:
   private String title;

   /** Constructs an instructor. */
   public Instructor(String name, String address,
                     String title) {
      super(name, address)
      this.title = title;

   }

   /** Textual description of this instructor. */
   @Override
   public String toString() {
      return title + " " + super.toString();
   }

   // More Instructor methods...
}
```

Typical `toString`

Displays the fields of the sub-class, and calls the `toString` of the super-class to display *its* fields.

Sub-class: Instructor

super-class

```
/** Represents a person. */
public class Person {

   // Person fields
 protected String name;
   protected String address;

   /** Constructs a Person */
 public Person(String name, String address) {
      this.name = name;
      this.address = address;
   }

   /** Sets the address of this person. */
 public void setAddress(String address) {
      this.address = address;
   }

   /** Textual description of this person. */
 public String toString() {
      name + ", " + address;
   }
   ...
}
```

sub-class

```
/** Represents an instructor. */
public class Instructor extends Person {
   // Instructor-specific fields:
   private String title;

   /** Constructs an instructor. */
   public Instructor(String name, String address,
                     String title) {
      super(name, address)
      this.title = title;

   }

   /** Textual description of this instructor. */
   @Override
   public String toString() {
      return title + " " + super.toString();
   }

   // More Instructor methods...
}
```

Alternative `toString` implementation:

```
   /** Textual description of this instructor. */
   @Override
   public String toString() {
      return title + " " + name + " " + address;
   }
```

Can you explain why it will work?

Overrides

super-class

```
/** Represents a person. */
public class Person {

   // Person fields
 protected String name;
   protected String address;

   /** Constructs a Person */
 public Person(String name, String address) {
      this.name = name;
      this.address = address;
   }

   /** Sets the address of this person. */
 public void setAddress(String address) {
      this.address = address;
   }

   /** Textual description of this person. */
 public String toString() {
      name + ", " + address;
   }
   ...
}
```

overrides

sub-class

```
/** Represents an instructor. */
public class Instructor extends Person {
   // Instructor-specific fields:
   private String title;

   /** Constructs an instructor. */
   public Instructor(String name, String address,
                     String title) {
      super(name, address)
      this.title = title;

   }

   /** Textual description of this instructor. */
   @Override
   public String toString() {
      return title + " " + super.toString();
   }

   // More Instructor methods...
}
```

A sub-class can *override* any super-class method;

(Not to be confused with *overloading*).

Inheritance in action

super-class

```
/** Represents a person. */
public class Person {

   // Person fields
 protected String name;
   protected String address;

   /** Constructs a Person */
 public Person(String name, String address) {
      this.name = name;
      this.address = address;
   }

   /** Sets the address of this person. */
 public void setAddress(String address) {
      this.address = address;
   }

   /** Textual description of this person. */
 public String toString() {
      name + ", " + address;
   }
   ...
}
```

sub-class

```
/** Represents an instructor. */
public class Instructor extends Person {
   // Instructor-specific fields:
   private String title;

   /** Constructs an instructor. */
   public Instructor(String name, String address,
                     String title) {
      super(name, address)
      this.title = title;

   }

   /** Textual description of this instructor. */
   @Override
   public String toString() {
      return title + " " + super.toString();
   }
   ...
```

client code

```
Instructor soc =
      new Instructor("Socrates", "Athens", "Prof.");
Instructor dick =
      new Instructor("Feynman", "LA", "Dr.");

soc.setAddress("Tel Aviv"); // changing soc's address

System.out.println(soc);
System.out.println(dick);
```

output

```
Prof. Socrates, Tel Aviv
Dr. Feynman, LA
```

Inheritance in action

super-class

```
/** Represents a person. */
public class Person {

   // Person fields
  protected String name;
    protected String address;

   /** Constructs a Person */
  public Person(String name, String address) {
       this.name = name;
       this.address = address;
    }

   /** Sets the address of this person. */
  public void setAddress(String address) {
       this.address = address;
    }

   /** Textual description of this person. */
  public String toString() {
       name + ", " + address;
```

sub-class

```
/** Represents an instructor. */
public class Instructor extends Person {
   // Instructor-specific fields:
   private String title;

   /** Constructs an instructor. */
   public Instructor(String name, String address,
                     String title) {
      super(name, address)
      this.title = title;

   }

   /** Textual description of this instructor. */
   @Override
   public String toString() {
      return title + " " + super.toString();
   }
   ...
```

client code

```
Instructor soc =
      new Instructor("Socrates", "Athens", "Prof.");
Instructor dick =
      new Instructor("Feynman", "LA", "Dr.");
soc.setAddress("Tel Aviv"); // changing soc's address
System.out.println(soc);
System.out.println(dick);
```

output

```
Prof. Socrates, Tel Aviv
Dr. Feynman, LA
```

Calling a super-class (super type) method

`soc` is an `Instructor`;

But `soc` is also a `Person`;

That's why we can call a `Person` method on an `Instructor` object.

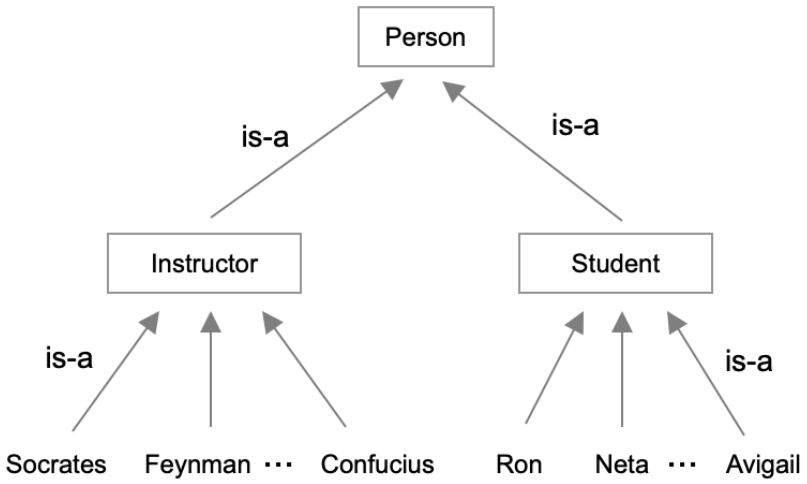
Sub-class: Student

Like any other sub-class:

Inherits all the super-class members;

Typically adds some members of its own;

Typically overrides some super-class method.



sub-class

```
/** Represents a student. */
public class Student extends Person {

    // Instructor-specific fields:
    private int studentId;
    private String club;

    /** Constructs a student. */
    public Student(String name, String address,
                   int studentId, String club) {
        super(name, address);
        this.studentId = studentId;
        this.club = club;
    }

    /** Textual description of this student. */
    @Override
    public String toString() {
        return super.toString() + ", id: " +
               studentId + ", club: " + club;
    }
    ...
}
```

Possible extensions (examples)

- List of courses, and grades
- GPA

...

Abstract class / abstract method

```
/** Represents a person. */
public abstract class Person {

   // Person fields
  protected String name;
    protected String address;

   /** Constructs a Person */
  public Person(String name, String address) {
      this.name = name;
      this.address = address;
    }

   /** Sets the address of this person. */
  public void setAddress(String address) {
      this.address = address;
    }

   /** Gets the role of this person. */
   // To be implemented by sub-classes.
  public abstract String getRole();

   /** Textual description of this person. */
  public String toString() {
      return getRole() + " " + name +
             ", " + address;
    }

   ...
}
```

Sometimes we want to *force* each sub-class to have some feature of its own.

Example: We want each instructor and student to have a *role* in the university.

Abstract method

A placeholder method, must be implemented by every sub-class of this class.

Technical point: If a class contains one or more *abstract methods*, it becomes an *abstract class*

Abstract class / abstract method

super-class

```
/** Represents a person. */
public abstract class Person {

   // Person fields
 protected String name;
   protected String address;

   /** Constructs a Person */
 public Person(String name, String address) {
      this.name = name;
      this.address = address;
   }

   /** Sets the address of this person. */
 public void setAddress(String address) {
      this.address = address;
   }

   /** Gets the role of this person. */
   // To be implemented by sub-classes.
 public abstract String getRole();

   /** Textual description of this person. */
 public String toString() {
      return getRole() + " " + name +
             ", " + address;
   }

   ...
}
```

must

sub-class

```
/** Represents an instructor. */
public class Instructor extends Person {
   // Instructor-specific fields:
   private String title;

/** Constructs an instructor. */
   public Instructor(String name, String address,
                     String title) {
      super(name, address)
      this.title = title;

   }

   /** Gets the role of this instructor. */
   @Override
   public String getRole() {
      return "Instructor " + title;

   }

   /** Textual description of this instructor. */
   @Override
   public String toString() {
      return super.toString();
   }
}
```

A sub-class *must* override every super-class *abstract* method;

A sub-class *can* override any other super-class method.

Abstract class / abstract method

super-class

```
/** Represents a person. */
public abstract class Person {

   // Person fields
  protected String name;
    protected String address;

   /** Constructs a Person */
  public Person(String name, String address) {
      this.name = name;
      this.address = address;
    }

   /** Sets the address of this person. */
  public void setAddress(String address) {
      this.address = address;
    }

   /** Gets the role of this person. */
   // To be implemented by sub-classes.
  public abstract String getRole();

   /** Textual description of this person. */
  public String toString() {
      return getRole() + " " + name +
             ", " + address;
    }

   ...
}
```

must

sub-class

```
/** Represents a student. */
public class Student extends Person {

    // Instructor-specific fields:
    private int studentId;
    private String club;

    /** Constructs a student. */
    public Student(String name, String address,
                   int studentId, String club) {
        super(name, address);
        this.studentId = studentId;
        this.club = club;
    }

    /** Gets the role of this students. */
    @Override
    public String getRole() {
        return "Student";
    }

    /** Textual description of this student. */
    @Override
    public String toString() {
        return super.toString() + ", id: " +
               studentId + ", club: " + club;
    }
    ...
}
```

A sub-class *must* override every super-class *abstract* method;

A sub-class *can* override any super-class method.

Inheritance in action

client code

```
// Creates some persons
Person[] persons = new Person[] {
   new Instructor("Socrates", "Raanana", "Prof."),
   new Instructor("Feynman", "Tel Aviv", "Dr."),
   new Student("Ron", "Raanana", 1001, "Dance"),
   new Student("Neta", "Raanana", 1002, "Debate"),
   new Student("Avigail", "Tel Aviv", 1003, "Dance")
};

// Prints all persons
for (Person person : persons) {
   System.out.println(person);
}
```

```
Instructor Prof. Socrates, Raanana
Instructor Dr. Feynman, Tel Aviv
Student Ron, Raanana, id: 1001, club: Dance
Student Neta, Raanana, id: 1002, club: Debate
Student Avigail, Tel Aviv, id: 1003, club: Dance
```

Polymorphism

Iterating over super-type objects, letting each object take care of itself (according to its sub-type).

Inheritance in action

client code

```
// Creates some persons
Person[] persons = new Person[] {
   new Instructor("Socrates", "Raanana", "Prof."),
   new Instructor("Feynman", "Tel Aviv", "Dr."),
   new Student("Ron", "Raanana", 1001, "Dance"),
   new Student("Neta", "Raanana", 1002, "Debate"),
   new Student("Avigail", "Tel Aviv", 1003, "Dance")
};

// Prints all persons
for (Person person : persons) {
   System.out.println(person);
}

// Prints all persons from Raanana
printPersonsFromCity(persons, "Raanana");

// Prints all the persons from the given city.
public static void printPersonsFromCity(Person[] persons, String city) {
   for (Person person : persons) {
      if (person.address.equals(city)) {
         System.out.println(person);
      }
   }
}
```

```
Instructor Prof. Socrates, Raanana
Instructor Dr. Feynman, Tel Aviv
Student Ron, Raanana, id: 1001, club: Dance
Student Neta, Raanana, id: 1002, club: Debate
Student Avigail, Tel Aviv, id: 1003, club: Dance
```

```
Instructor Prof. Socrates, Raanana
Student Ron, Raanana, id: 1001, club: Dance
Student Neta, Raanana, id: 1002, club: Debate
```

Polymorphism

Iterating over super-type objects, letting each object take care of itself (according to its sub-type).

Lecture plan

- Inheritance / sub-classing
- Inheritance vs composition
- The class `Object`
- Important `Object` methods

Inheritance example: Colored Points

super-class

```
/** Represents a point. */
public class Point {

    // The coordinates of this point
    protected int x, y;

    /** Constructs a point. */
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    /** Textual representation */
    public String toString() {
        return "(" + x + "," + y + ")";
    }
    ...
}
```

sub-class

```
import java.awt.Color;

/** Represents a colored point. */
public class ColoredPoint extends Point {

    // The color of this point.
    private Color color;

    /** Constructs a colored point. */
    public ColoredPoint(int x, int y, Color color) {
        super(x,y); // calls a super constructor
        this.color = color;
    }

    /** Constructs a point, colored black */
    public ColoredPoint(int x, int y) {
        this(x, y, Color.black);
    }

    /** Textual representation of this point. */
    public String toString() {
        return super.toString() + " " + color;
    }
    ...
```

Client code:

```
// Creates and prints colored points:
ColoredPoint cp1 = new ColoredPoint(3,2,Color.red);
System.out.println(cp1);
ColoredPoint cp2 = new ColoredPoint(5,7);
System.out.println(cp2);
```

```
// Output:
(3,2) [r=255,g=0,b=0]
(5,7) [r=0,g=0,b=0]
```

return values of the `toString()` method of `Color`

Inheritance example: Colored Points

```
/** Represents a point. */
public class Point {

    // The coordinates of this point
    protected int x, y;

    /** Constructs a point. */
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    /** Textual representation */
    public String toString() {
        return "(" + x + "," + y + ")";
    }
    ...
}
```

super-class

```
import java.awt.Color;

/** Represents a colored point. */
public class ColoredPoint extends Point {

    // The color of this point.
    private Color color;
```

Sub-classing

Same code as previous slide

```
import java.awt.Color;

/** Represents a colored point. */
public class ColoredPoint {

    private Point point
    private Color color;

    /** Constructs a colored point. */
    public ColoredPoint(int x, int y, Color color) {
        this.point = new Point(x,y);
        this.color = color;
    }

    /** Textual representation of this point. */
    public String toString() {
        return point + " " + color;
    }
    ...
}
```

Composing

A regular class that has an object as one of its fields (**no inheritance**)

`ColoredPoint` objects are *composed* from `Point` objects

Best practice

Inheritance has advantages, but results in complex systems;

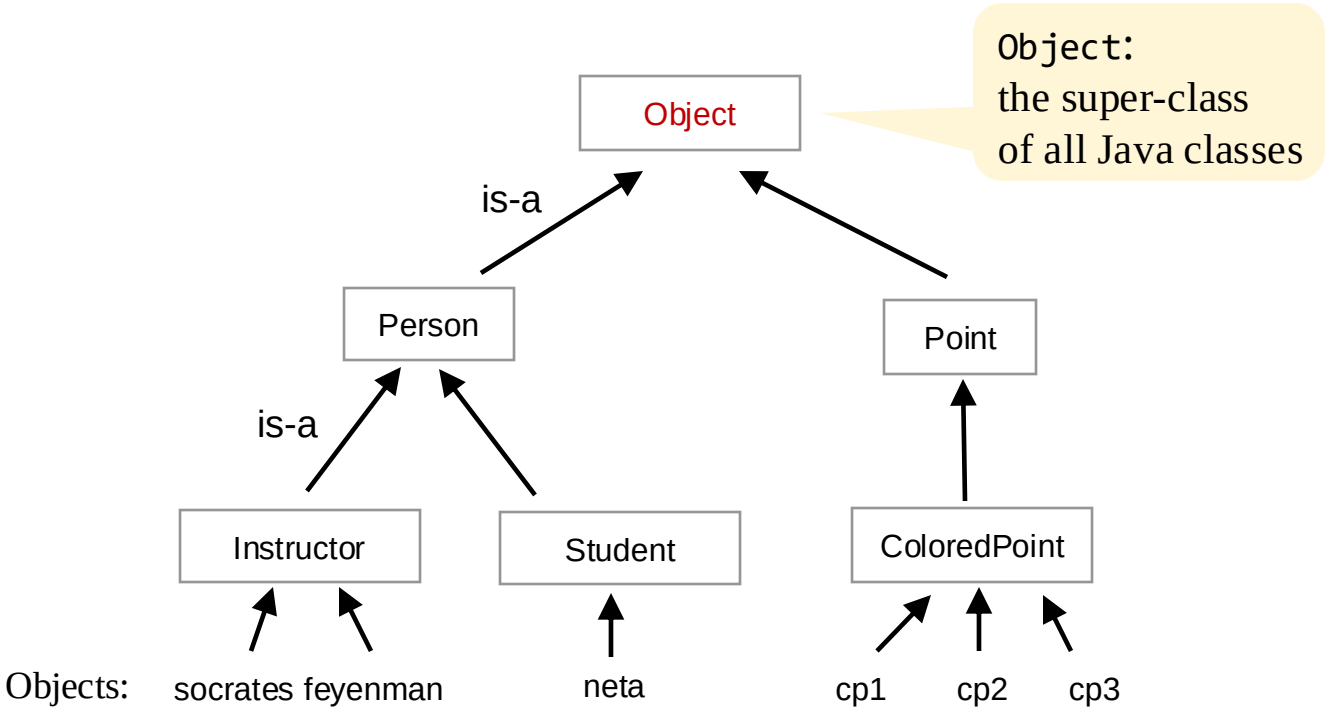
When possible, favor composition over inheritance.

(KISS)

Lecture plan

- Inheritance / sub-classing
- Inheritance vs composition
- The class `Object`
- Important `Object` methods

The class / type hierarchy



Every class in Java is a sub-class of the `Object` class;

Therefore, every Java object inherits all the `Object` methods;

Following this convention, Java's designers made the `Object` class declare a few *default methods* that they wanted every object to have.

`Object` methods

Class Object

java.lang.Object

`public class Object`

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass.

Method Summary

`protected Object`	`clone()`	Creates and returns a copy of this object.
`boolean`	`equals(Object obj)`	Indicates whether some other object is "equal to" this one.
`protected void`	`finalize()`	**Deprecated.** The finalization mechanism is inherently problematic.
`Class<?>`	`getClass()`	Returns the runtime class of this `Object`.
`int`	`hashCode()`	Returns a hash code value for the object.
`void`	`notify()`	Wakes up a single thread that is waiting on this object's monitor.
`void`	`notifyAll()`	Wakes up all threads that are waiting on this object's monitor.
`String`	`toString()`	Returns a string representation of the object.

`Object` methods

```
toString():          // Returns a textual representation of this object
```

- `equals(Object obj):` // Compares this object to another object
- `hashCode():` // Provides a unique ID for this object

Object methods: toString

super class of all objects

```
/** Represents every object in every class. */
public class Object {
    ...
    /** Returns the class of this object. */
    public Class getClass() { ... }

    /** Returns the hash code value of this object. */
    public int hashCode() { ... }

    ...

    /** Default toString implementation */
    public String toString() {
        return getClass().getName() + "@" +
               Integer.toHexString(hashCode());
    }
    // More Object methods
}
```

example of a class that has no toString method

```
/** Represents a point. */
public class Point {

    // The coordinates of this point
    private int x, y;

    /** Constructs a point. */
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // This Point class has no toString method.
    ...
}
```

```
// Client code, in some class:
Point p1 = new Point(2,3);
Point p2 = new Point(2,3);
System.out.println(p1);
System.out.println(p2);
```

```
// Output
Point@8fd9b4d
Point@f7b650a
```

Java calls the first `toString()` method up the class hierarchy. In this case, it's the `toString` of `Object`

`Object` methods: `toString`

super class of all objects

```
/** Represents every object in every class. */
public class Object {
    ...
    /** Returns the class of this object. */
    public Class getClass() { ... }
    /** Returns the hash code value of this object. */
    public int hashCode() { ... }
    ...
    /** Default toString implementation */
    public String toString() {
        return getClass().getName() + "@" +
               Integer.toHexString(hashCode());
    }
    // More Object methods
}
```

example of a class that has no `toString` method

```
/** Represents a point. */
public class Point {

    // The coordinates of this point
    private int x, y;

    /** Constructs a point. */
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // This Point class has no toString method.
    ...
}
```

```
// Client code, in some class:
Point p1 = new Point(2,3);
Point p2 = new Point(2,3);
System.out.println(p1);
System.out.println(p2);
```

```
// Output
Point@8fd9b4d
Point@f7b650a
```

`getClass` (an `Object` method) returns the class to which this object belongs; The class is returned as a `Class` object;

`getName` (a `Class` method) returns the class name;

`hashCode` (an `Object` method) returns the unique object ID.

`Object` methods: `toString`

super class of all objects

```
/** Represents every object in every class. */
public class Object {
    ...
    /** Returns the class of this object. */
    public Class getClass() { ... }
    /** Returns the hash code value of this object. */
    public int hashCode() { ... }
    ...
    /** Default toString implementation */
    public String toString() {
        return getClass().getName() + "@" +
               Integer.toHexString(hashCode());
    }
    // More Object methods
}
```

```
/** Represents a point. */
public class Point {

    // The coordinates of this point
    private int x, y;

    /** Constructs a point. */
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    /** A textual representation of this point. */
    public String toString() {
        return "(" + x + "," + y + ")";
    }

    ...
}
```

```
// Client code, in some class:
Point p1 = new Point(2,3);
Point p2 = new Point(2,3);
System.out.println(p1);
System.out.println(p2);
```

```
// Output
(2,3)
(2,3)
```

Best practice:

Always write a `toString()` method,
to override the default implementation.

<u>The result:</u> A sensible way to display `Point` objects.

`Object` methods

- `toString():` // Returns a textual representation of this object
- `equals(Object obj):` // Compares this object to another object
- `hashCode():` // Provides a unique ID for this object

`Object` methods: `equals`

super class of all objects

```
/** Represents every object in every class. */
public class Object {
    ...
    /** Returns the class of this object. */
    public Class getClass() { ... }

    /** Returns the hash code value of this object. */
    public int hashCode() { ... }

    ...

    /** Returns a textual representation of this object */
    public String toString() {
        return getClass().getName() + "@" +
               Integer.toHexString(hashCode());
    }

    /** Default "equals to" implementation. */
    public boolean equals(Object other) {
        return (this == other);
    }

    // More Object methods
}
```

example of a class that has no `equals` method

```
/** Represents a point. */
public class Point {

    // The coordinates of this point
    private int x, y;

    /** Constructs a point. */
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // This Point class has no equals method.
    ...
}
```

```
// Client code (in any class):
Point p1 = new Point(2,3);
Point p2 = new Point(2,3);
System.out.println(p1.equals(p2));  // false
```

The `equals()` method of the `Object` class performs the most basic equality check, ignoring the object's semantics.

Result: Objects that are semantically equal are considered unequal.

`Object` methods: `equals`

super class of all objects

```
/** Represents every object in every class. */
public class Object {
    ...
    /** Returns the class of this object. */
    public Class getClass() { ... }

    /** Returns the hash code value of this object. */
    public int hashCode() { ... }

    ...

    /** Returns a textual representation of this object */
    public String toString() {
        return getClass().getName() + "@" +
               Integer.toHexString(hashCode());
    }

    /** Default "equals to" implementation. */
    public boolean equals(Object other) {
        return (this == other);
    }

    // More Object methods
}
```

```
/** Represents a point. */
public class Point {

    // The coordinates of this point
    private int x, y;

    /** Constructs a point. */
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    /** Checks if this point equals to the other point. */
    public boolean equals(Point other) {
        return ((x == other.x) &&
                (y == other.y));
    }
    ...
}
```

```
// Client code (in any class):
Point p1 = new Point(2,3);
Point p2 = new Point(2,3);
System.out.println(p1.equals(p2));  // true
```

Best practice:

Always write an `equals` method, to override the default implementation.

Result: A semantically sensible way to check if two objects are equal.

Equals (industrial-strength version...)

super class of all objects

```
/** Represents every object in every class. */
public class Object {
    ...
    /** Returns the class of this object. */
    public Class getClass() { ... }
    /** Returns the hash code value of this object. */
    public int hashCode() { ... }
    ...
    /** Returns a textual representation of this object */
    public String toString() {
        return getClass().getName() + "@" +
               Integer.toHexString(hashCode());
    }
    /** Default "equals to" implementation. */
    public boolean equals(Object other) {
        return (this == other);
    }
    // More Object methods
}
```

The `@Override` tag: tells the compiler to verify that the signature of this method is identical to the signature of the super method.

Best practice: Use this tag when overriding a method.

```
/** Represents a point. */
public class Point {
    // The coordinates of this point
    private int x, y;
    ...
    /** Checks if this point equals to the other point.
    @Override
    public boolean equals(Object other) {
       // A complete, bullet-proof implementation of "equals" :
       if (other == null) return false;
       if (this == other) return true;
       // Note: In Java, during runtime, objects are allowed to
       // change their types (discussed later...). Therefore, when
       // clients call obj1.equals(obj2), we must check
       // that the two objects have the same type:
       if (getClass() != other.getClass()) return false;
       // Now we know that other is a Point object,
       // but the compiler doesn't know it. So, we cast it:
       Point other = (Point) other;
       // And only now we can safely check all the fields:
       if (x != other.x) return false;
       if (y != other.y) return false;
       return true;
       // Sigh!
    }
    // More Point methods
}
```

`Object` methods

- `toString():` // Returns a textual representation of this object
- `equals(Object obj):` // Compares this object to another object
- `hashCode():` // Provides a unique ID for this object

`Object` methods: `hashCode`

super class of all objects

```
/** Represents every Object. */
public class Object {
    ...
    /** Checks if this object "equals to" the other object. */
    public boolean equals(Object other) {
        return (this == other);
    }
    /** Default hashCode implementation. */
    public int hashCode() {
        // Objects that are physically different get
        // different hash codes. Code omitted.
    }
    // More Object methods
}
```

example of a class that has no `hashCode` method

```
/** Represents a point. */
public class Point {
    // The coordinates of this point
    private int x, y;
    ...
    /** Checks if this point equals to the other point. */
    public boolean equals(Point other) {
        return ((x == other.x) && (y == other.y));
    }
    // This Point class has no hashCode method.
    ...
}
```

Hash code contract

$$\texttt{equals(obj1, obj2)} \leftrightarrow \texttt{h(obj1)} = \texttt{h(obj2)}$$

$$\texttt{!equals(obj1, obj2)} \leftrightarrow \texttt{h(obj1)} \neq \texttt{h(obj2)}$$

```
// Client code, in some class:
Point p1 = new Point(2,3);
Point p2 = new Point(2,3);
Point p3 = new Point(1,2);
System.out.println(p1.hashCode()); // 150838093
System.out.println(p2.hashCode()); // 259745034
System.out.println(p3.hashCode()); // 205146899
```

the contract is broken

`Object` methods: `hashCode`

super class of all objects

```
/** Represents every Object. */
public class Object {
    ...
    /** Checks if this object "equals to" the other object. */
    public boolean equals(Object other) {
        return (this == other);
    }
    /** Default hashCode implementation. */
    public int hashCode() {
        // Objects that are physically different get
        // different hash codes. Code omitted.
    }
    // More Object methods
}
```

```
/** Represents a point. */
public class Point {
    // The coordinates of this point
    private int x, y;
    ...
    /** Checks if this point equals to the other point. */
    public boolean equals(Point other) {
        return ((x == other.x) && (y == other.y));
    }
    /** Returns the hash code of this point. */
    public int hashCode() {
        int prime = 31;
        int result = 1;
        result = prime * result + x;
        result = prime * result + y;
        return result;
    }
    ...
}
```

Best practice:

Write a `hashCode()` method,
to override the default implementation.

Hash code contract

`equals(obj1, obj2)` ↔ `h(obj1) = h(obj2)`

`!equals(obj1, obj2)` ↔ `h(obj1) ≠ h(obj2)`

```
// Client code, in some class:
Point p1 = new Point(2,3);
Point p2 = new Point(2,3);
Point p3 = new Point(1,2);
System.out.println(p1.hashCode()); // 1026
System.out.println(p2.hashCode()); // 1026
System.out.println(p3.hashCode()); // 994
```

The contract is maintained

`Object` methods

- `toString():` // Returns a textual representation of this object
- `equals(Object obj):` // Compares this object to another object
- `hashCode():` // Provides a unique ID for this object

Best practice

When writing a class (for real...), always implement these 3 methods;

Why? These methods play important roles in many programming settings, and other classes (that you don't control) assume that you've implemented them;

Good news: IDEs offer automatic / default creation of these methods.

Recap: Inheritance

Reasons to learn class inheritance

- Many software systems are based on inheritance
- The Java class library is based on inheritance (and so is its documentation)
- You may have to develop / debug / extend systems that are based on inheritance

Observations

- **Class inheritance** (this lecture) is powerful, but complex and error-prone
- **Interface inheritance** (next lecture) is powerful, and *simple*.