Lecture 10-2

Inheritance

Inheritance lectures outline

Lecture 10-1:

Interfaces
- Motivation
- Examples

Inheritance
- Motivation
- Examples

Sub-classing
- Constructors
- Methods

Lecture 10-2:

Narrowing / widening

Class Object
- `toString`
- `Equals`
- `HashCode`

Run-time types

Virtual methods

Visibility

The `final` modifier

The `Comparable` interface

A typical inheritance hierarchy

base-class

```
// Represents a point on a grid.
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;
  }

  // Computes the distance from another point
  public double distanceFrom (Point p) {
    int dx = x - p.x;
    int dy = y - p.y;
    return Math.sqrt(dx * dx + dy * dy);
  }

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

subclass

```
// Represents a colored point on a grid.
public class ColoredPoint extends Point {

    private Color color;

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

enum Color {red, blue, green, yellow};
```

client

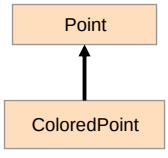
```
Point p = new Point(50,100);
ColoredPoint cp = new ColoredPoint(10,20,Color.red);
System.out.println(cp);  // prints (10,20)
```


The `instanceof` operator

Problem: How to tell if some object variable, say `obj`, points to an object of type `SomeClass`?

Solution: Check if `obj instanceof SomeClass`

```
// Recall that ColoredPoint extends Point
Point p = new Point(50,100);
ColoredPoint cp = new ColoredPoint(10,20,Color.red);
String s = new String("bla");
System.out.println (p instanceof Point);          // true
System.out.println (cp instanceof ColoredPoint);  // true
System.out.println (cp instanceof Point);         // true
System.out.println (p instanceof ColoredPoint);   // false
System.out.println (s instanceof Point);          // compilation error
```



When using `obj instanceof SomeClass`, `obj` must be "related" to `SomeClass`.

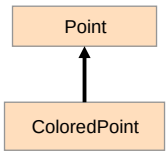
If an object is an instance of some class, it is also an instance of it's base-class.

Narrowing / widening

Sometimes we want to "widen-up" an object and treat it like an instance of its base class

In other times we want to "narrow down" an object and treat it like an instance of one of its subclasses

```
Point p = new Point(10,20);
Point cp = new ColoredPoint(2,3,Color.red);
p = (Point) cp;          // cp is widened up (explicitly)
p = cp;                  // cp is widened up (implicitly)
cp = (ColoredPoint) p ;  // p is narrowed down
cp = new Point(10,20);
```



In Java:

- Widening up can be done either explicitly or implicitly
- Narrowing down requires explicit casting.

Implicit widening by parameter passing

base-class

```
// A point on a grid
public class Point {
  private int x, y;
  ...
  // Computes the distance from another point
  public double distanceFrom (Point p) {
    int dx = x - p.x;
    int dy = y - p.y;
    return Math.sqrt(dx * dx + dy * dy);
  }
  ...
}
```

subclass

```
// Represents a colored point on a grid.
public class ColoredPoint extends Point {
  private Color color;
  ...
}

enum Color {red, blue, green, yellow};
```

Client

```
Point p = new Point(2,3);
ColoredPoint cp = new ColoredPoint(5,6,Color.red);
double d = p.distanceFrom(cp);
```

The method expects to get a `Point` parameter; Instead, it gets a `ColoredPoint` argument (`cp`).

That's OK – `cp` is widened up by the type of the formal parameter.

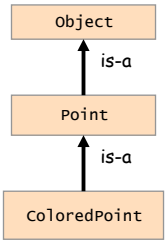
The inheritance principle: a subclass object can be used wherever an object from its base-class is expected.

Method calling up the inheritance hierarchy

When a method is called on some object:

- If the compiler finds a matching method declaration in the object's class, it uses it
- Otherwise it searches the method in the immediate parent class
- All the way up to `java.lang.Object`

(An example is shown in slide 3, when we invoked `toString` on a `ColoredPoint` object that has no `toString` implementation)



Outline

Lecture 10-1:

Interfaces
- Motivation
- Examples

Inheritance
- Motivation
- Examples

Sub-classing
- Constructors
- Methods

Lecture 10-2:

Narrowing / widening

Class Object
- `toString`
- `Equals`
- `HashCode`

Run-time types

Virtual methods

Visibility

The `final` modifier

The `Comparable` interface

`Class Object`: the parent class of all Java classes

public class **Object**

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

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**() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class	**getClass**() Returns the runtime class of an 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.

(partial API) ⇨ : Discussed in this lecture

- Class `object` is the root of the class hierarchy
- Every Java class implicitly extends `Object`
- Thus, the `Object` methods are inherited by all Java classes
- Class designers are expected to override some of these base methods.

The `toString` method

toString (`java.lang.Object` API)

`public String toString()`

Returns a string representation of the object. In general, the `toString` method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.

The `toString` method for class `Object` returns a string consisting of the name of the class of which the object is an instance, the at-sign character `'@'` and the unsigned hexadecimal representation of the hash code of the object.

When you override a method, you are expected to follow some rules

If the base-class is documented properly, these rules should be stated in its API (as seen above).

Overriding `toString`

base-class

```
// Represents a point on a grid.
public class Point {

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

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

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

overriding

- Note the difference between method overriding and overloading

subclass

```
// Represents a colored point on a grid.
public class ColoredPoint extends Point {

  private Color color;

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

  public String toString () {
    return super.toString() + " "
           + color.toString();
  }
  . . .
}

enum Color {red, blue, green, yellow};
```

Overridden methods can be invoked using the syntax `super.methodName`

The `equals` method

equals

`public boolean equals(Object obj)`

The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same object (`x == y` has the value `true`).

- Class designers normally override `equals()`, to reflect an equality relationship that makes sense given this class semantics
- Important: if you override `equals()` you must also override `hashCode()`

  (more about `hashCode` later).

The `equals` method

```
public class Point {
    // The coordinates of the point
    private int x, y;
    ...
    public boolean equals (Point other) {
        if (other == this) return true;
        if (other == null) return false;
        return (x == other.x && y == other.y);
    }
   ...
}
```

client

```
Point p1 = new Point(10,20);
Point p2 = new Point(10,20);

System.out.println(p1 == p2);       // false

System.out.println(p1.equals(p2));  // ?

// If equals was not overridden by Point, we'll get false.
// If it was overridden by Point properly, we'll get true.
```


Overriding the `equals` method

base class

```
public class Point {
  // The coordinates of the point
  private int x, y;
  ...
  public boolean equals (Object obj) {
     if (obj == this) return true;
     if (obj == null) return false;
     if (!(obj instanceof Point)) return false;
     Point other = (Point) obj;
     return (x == other.x && y == other.y);
    }
  ...
}
```

A more general version of `equals`, designed to accommodate `equals` calls from subclasses

subclass

```
public class ColoredPoint extends Point {
  private Color color;
  ...
  public boolean equals (ColoredPoint other) {
     if (this == other) return true;
     if (super.equals(other) && (color == other.color)) return true;
     return false;
  }
  ...
}
```

Overriding the `equals` method, final version

base class

```
public class Point {
  // The coordinates of the point
  private int x, y;
  ...
  public boolean equals (Object obj) {
     if (this == obj) return true;
     if (obj == null) return false;
     if (!(obj instanceof Point)) return false;
     Point other = (Point) obj;
     return (x == other.x && y == othe
   }
  ...
}
```

These `equals` methods were generated automatically by Eclipse.

subclass

```
public class ColoredPoint extends Point {
  private Color color;
  ...
  public boolean equals (Object obj) {
     if (this == obj) return true;
     if (!super.equals(obj)) return false;
     if (!(obj instanceof ColoredPoint)) return false;
     ColoredPoint other = (ColoredPoint) obj;
     if (color != other.color) return false;
     return true;
  }
}
```

Likewise, designed to accommodate `equals` calls from subclasses of `ColoredPoint`, if and when such subclasses will be defined.

Overriding `hashCode`

```
public class Point {
    // The coordinates of the point
    private int x, y;
    ...
    public int hashCode () {
        final int prime = 31;
        int result = 1;
        result = prime * result + x;
        result = prime * result + y;
        return result;
    }
  ...
}
```

- Every Java object has a numeric unique identifier, called "hash code"
- The `Object` class has a `hashCode` method that returns this number (that's the number that the standard `toString` method returns)
- Overriding implementations of `hashCode` are expected to create an implementation that ensures that `obj1.hashCode() = obj2.hashCode()` if and only if `obj1.equals(obj2)`
- This can be done by implementing a certain function on the object's field values
- The particular `hashCode` method shown on the left was generated automatically by Eclipse.

Overriding rules

- A subclass can override methods from its base-class
- The base-class API should state what is expected of overriding implementations of its methods
- The method signatures (name, number and type of the parameters) of the overriding method and the overridden method must be identical
- The visibility modifier of the overriding method can allow more access than the overridden method, but not less. For example, a `protected` method in the base-class can be made `public` but not `private`.
- The overriding method can have a different `throws` clause as long as it doesn't declare any types not declared by the `throws` clause in the overridden method.

Outline

Lecture 10-1:

Interfaces
- Motivation
- Examples

Inheritance
- Motivation
- Examples

Sub-classing
- Constructors
- Methods

Lecture 10-2:

Narrowing / widening

Class Object
- `toString`
- `Equals`
- `HashCode`

Run-time types

Virtual methods

Visibility

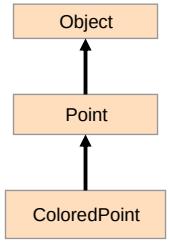
The `final` modifier

The `Comparable` interface

Run-time type

Consider the following code:

```
Point p;
ColoredPoint cp = new ColoredPoint(10, 20, Color.green);
p = cp;
```



Terminology: We say that …

The *compile-time* type of `p` is `Point`

The *run-time type* of `p` is `ColoredPoint`

Run-time type

- The run-time type is always some subclass of the compile-time type
- The same object can have different run-time types in different program runs.

Virtual method calling

- In an object-oriented language, (non-static) methods are always invoked on some object
- If the object has been narrowed or widened, which method implementation should be used?

Virtual method calling:
The method to be invoked is determined by the run-time type of the object (not by its compile-time type)

In Java, all methods invocations are virtual (unlike C++).

base-class

```
public class Animal {
  public String eats() { return "food"; }
}
```

subclass

```
public class Cow extends Animal {
  public String eats() { return "grass"; }
}
```

Client

```
// Narrowing demo
Animal a;
Cow c = new Cow();
a = c;
System.out.println(a.eats());
// prints “grass”
```

Outline

Lecture 10-1:

Interfaces
- Motivation
- Examples

Inheritance
- Motivation
- Examples

Sub-classing
- Constructors
- Methods

Lecture 10-2:

Narrowing / widening

Class Object
- `toString`
- `Equals`
- `HashCode`

Run-time types

Virtual methods

Visibility

The `final` modifier

The `Comparable` interface

Visibility modifiers

The visibility modifier of a member determines which other classes can access the member.

There are four possibilities:

`private` : Accessible to this class only

`public` : Accessible to any class

`protected` : (1) Accessible to any class in the same package as this class
(2) Accessible to any subclass of this class

None (default): Package-private: accessible to any class in the same package as this class

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Usage

- Use `protected` to expose a member to subclasses and hide it from the rest of the world
- Avoid defining too many protected variables: it hurts encapsulation
- A `protected` member is considered part of the class interface and should be documented in the class API.

The `final` modifier

The `final` modifier can be applied to classes, methods and variables; in each case it has a different meaning:

- `final className` indicates that the class cannot have subclasses
- `final methodName` indicates that the method cannot be overridden
- `final variableName` indicates that the variable can be initialized only once

Declaring a class `final` can improve performance, since Java does not have to maintain the run-time types of its objects.

Case in point: the `String` class is declared `final`.

Example of a final method

`Class Object` features a `getClass` method that returns the run-time type of any given object. For example:

```
Point cp = new ColoredPoint(2,3,Color.red);
System.out.println(cp.getClass()); // prints "class ColoredPoint“
cp = new Point(10,20);
System.out.println(cp.getClass()); // prints "class Point"
```

Many classes and programmers expect `getClass` to work that way.

Therefore, letting other classes override it makes no sense.

To prevent overriding, this method is declared (in `Class Object`) as `final`.

Outline

Lecture 10-1:

Interfaces
- Motivation
- Examples

Inheritance
- Motivation
- Examples

Sub-classing
- Constructors
- Methods

Lecture 10-2:

Narrowing / widening

Class Object
- `toString`
- `Equals`
- `HashCode`

Run-time types

Virtual methods

Visibility

The `final` modifier

The `Comparable` interface

The `Comparable` interface

- Value classes (like `Point`, `Time`) represent objects that typically have a natural order
- Therefore, it makes sense that these classes will offer a comparison service
- In Java, object comparisons are standardized by the `Comparable` interface:

`Interface Comparable`:

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method.

Lists and arrays of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort). Objects that implement this interface can be used as keys in a sorted map or elements in a sorted set, without the need to specify a comparator.

public int **compareTo** (Object o)

Parameters: o - the Object to be compared.

Returns: a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Throws: ClassCastException - if the specified object's type prevents it from being compared to this Object.

(from the `java.lang.Comprabale` API)

Using `Comparable`

```
// Represents a point on a grid.
public class Point implements Comparable {
    private int x, y;
    ...
    // We say that Point p is "greater than"
    // point q if p.x + p.y > q.x + q.y
    public int compareTo (Object obj) {
       Point other = (Point) obj;
       if ((x + y) == (other.x + other.y))
          return 0;
       if ((x + y) < (other.x + other.y))
          return -1;
       return 1;
    }
    ...
}
```

By implementing `Comparable`, a class indicates that its instances have a natural ordering

For a small price, you gain significant benefits:

`Comparable` objects can be sorted and used in numerous Java collections that depend on order.

client

```
Point[] points = new Point[5];
Random rnd = new Random();
for (int i = 0; i < points.length; i++)
    points[i] = new Point(rnd.nextInt(10) , rnd.nextInt(10));
for (Point p : points) System.out.print(p + " ");

Arrays.sort(points);
System.out.println();
for (Point p : points) System.out.print(p + " ");
```

Output:

```
(8,2) (4,4) (9,7) (6,2) (6,0)
(6,0) (4,4) (6,2) (8,2) (9,7)
```