

CS1110 Classes, wrapper classes, Vectors. 10 Feb 2012

Miscellaneous points about classes.
Discussion of wrapper classes and class Vector



Use the text as a reference.

1. Want to know about type **int**? Look it up in text.
2. Want to know about packages? Look up "packages".
3. How is the new-exp evaluated? Look it up.
4. etc.

Content of this lecture

Go over miscellaneous points to round out your knowledge of classes and subclasses. There are a few more things to learn after this, but we will handle them much later.

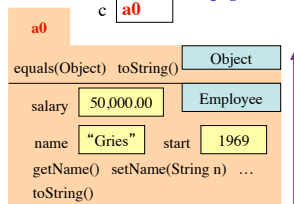
- Inheriting fields and methods and overriding methods. Sec. 4.1 and 4.1.1: pp. 142–145
- Purpose of **super** and **this**. Sec. 4.1.1, pp. 144–145.
- More than one constructor in a class; another use of **this**. Sec. 3.1.3, pp. 110–112.
- Constructors in a subclass —calling a constructor of the super-class; another use of **super**. Sec. 4.1.3, pp. 147–148.
- Wrapper classes. Read Section 5.1.

Employee c = new Employee("Gries", 1969, 50000);
c.toString() Sec. 4.1, page 142

Which method toString() is called?

Overriding rule, or bottom-up rule:

To find out which is used, start at the bottom of the class and search upward until a matching one is found.



This class is on page 105 of the text.

Terminology. Employee inherits methods and fields from Object. Employee overrides function toString.

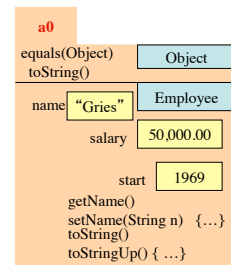
Purpose of super and this Sec. 4.1, pages 144-145
this refers to the name of the object in which it appears.
super is similar but refers only to components in the partitions above.

```
/** = String representation of this Employee */
public String toString() {
    return this.getName() + ", year " +
        getStart() + ", salary " + salary;
}
```

ok, but unnecessary

```
/** = name of this object */
public String toStringUp() {
    return super.toString();
}
```

necessary



A second constructor in Employee Sec. 3.1.3, page 110
Provide flexibility, ease of use, to user

```
/** Constructor: a person with name n, year hired d, salary s */
public Employee(String n, int d, double s) {
    name = n; start = d; salary = s;
} // First constructor
```

```
/** Constructor: a person with name n, year hired d, salary 50,000 */
public Employee(String n, int d) {
    name = n; start = d; salary = 50000;
} // Second constructor; salary is always 50,000
```

```
/** Constructor: a person with name n, year hired d, salary 50,000 */
public Employee(String n, int d) {
    this(n, d, 50000);
} // Another version of second constructor; calls first constructor
```

Here, this refers to the other constructor. You HAVE to do it this way

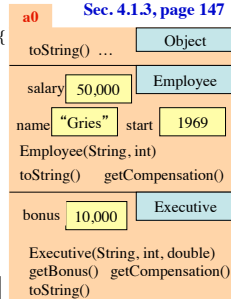
```
public class Executive extends Employee {
    private double bonus;
    /** Constructor: name n, year hired d, salary 50,000, bonus b */
    public Executive(String n, int d, double b) {
        super(n, d);
        bonus = b;
    }
}
```

The first (and only the first) statement in a constructor has to be a call on another constructor. If you don't put one in, then this one is automatically used:

```
super();
```

Principle: Fill in superclass fields first.

Calling a superclass constructor from the subclass constructor Sec. 4.1.3, page 147



```

public class Executive extends Employee {
    public Executive(String n, int d, double b) {
        bonus = b;
    }
}

```

One constructor in Employee

First statement in constructor: constructor call. If none, Java inserts:

```

super();

```

Is above program okay?

A. Compiles with no change
 B. Compiles with **super()** inserted
 C. Doesn't compile

Class hierarchy diagram:

- Object
 - Employee
 - Executive
 - salary: 50,000
 - name: "Gries" start: 1969
 - bonus: 10,000

Executive methods: Executive(String, int, double), getBonus(), getCompensation(), toString()

Employee methods: toString() ...

7

Wrapper classes.

Read Section 5.1

Integer

```

Integer(int) Integer(String)
toString() equals(Object) intValue()

```

Static components:
 MIN_VALUE MAX_VALUE
 toString(int) toBinary(int)
 valueOf(String) parseInt(String)

Soon, we'll wish to deal with an **int** value as an object.


"Wrapper class" Integer provides this capability.

An instance of class Integer contains, or "wraps", one **int** value.


Can't change the value. *immutable*.

Instance methods: constructors, toString(), equals, intValue.


Static components provide important extra help.




What is a wrapper?
 Something that holds another thing
 -wraps around it



Sandwich wrapper



Spring rolls wrappers



cupcake wrapper

an **int** wrapper

Integer

5



wriggle wrapper

10

Each primitive type has a corresponding wrapper class. When you want to treat a primitive value of that type as an object, then just wrap the primitive value in an object of the wrapper class!

Primitive type	Wrapper class
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Each wrapper class has:

- Instance methods, e.g. equals, constructors, toString,
- Useful static constants and methods.

```

Integer k = new Integer(63);    int j = k.intValue();

```

You don't have to memorize the methods of the wrapper classes. But be aware of them and look them up when necessary. Use Gries/Gries, Section 5.1, and ProgramLive, 5-1 and 5-2, as references.

11

Class Vector

An instance of class Vector maintains an expandable/shrinkable list of objects. Use it whenever you need to maintain a list of things.

Values of primitive types cannot be placed directly into the list of a Vector. That's why we have the wrapper classes. In the interactions pane, we will do a few things, like this:

```

import java.util.*;
Vector v = new Vector();
v.add(new Integer(2));
v.add(3);
v.add('c');

```

In newer versions of Java, v.add(1) is allowed; the 1 is wrapped in an Integer object and the name of that object is added to v.
 Doesn't work in older versions.

12