

Java Programming

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

- Overloading
- Polymorphism
- Overriding

Today's Lecture

- We will start with method signatures overloading...

Method Signatures and Overloading

- Signatures identify methods.
- Method **signature** consists of two pieces:
1. Method name
2. Method parameters
- Method signatures must be unique within a given scope (for example inside a class).
- Cannot have two methods with the same signature ***in the same scope***.
- Return type is NOT part of the signature!

Method Signature

- What are the method signatures?

```
public class Test
{
    public void H() { System.out.println("Hello"); }
    public void G() { System.out.println("Goodbye"); }
    public void I(int num) { System.out.println(num); }
    public void J(String s, int num) {
        System.out.printf("%s %d\n", s, num);
    }
    public void K(int num, String s) {
        System.out.printf("%s %d\n", s, num);
    }
}
```

- The method signatures are:

<u>Signature</u>	<u>Name</u>	<u>Parameters</u>
H()	H	none
G()	G	none
I(int num)	I	int
J(String s, int num)	J	String, int
K(int num, string s)	K	int, String

- **Is this legal? Are methods ambiguous?**

```
public class Test
{
    public void H()
    {
        System.out.println("Hello");
    }

    public void G() {
        System.out.println("Goodbye");
    }
}
```

- **YES. It is legal.**

```
public class Test
```

```
{
```

```
    public void H()
```

```
    {
```

```
        System.out.println("Hello");
```

```
    }
```

```
    public void G() {
```

```
        System.out.println("Goodbye");
```

```
    }
```

```
}
```

**LEGAL. Same
parameter lists
but different
names so OK.**



- **Is this legal?**

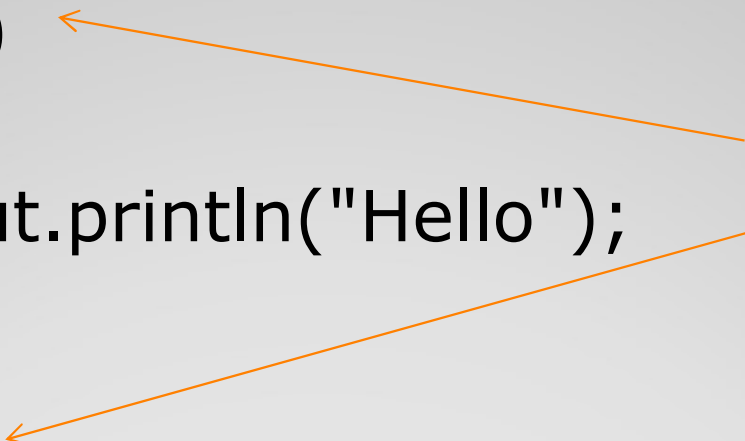
```
public class Test
{
    public void H()
    {
        System.out.println("Hello");
    }

    public void H() {
        System.out.println("Goodbye");
    }
}
```

- **NO. It is not legal.**

```
public class Test
{
    public void H()
    {
        System.out.println("Hello");
    }

    public void H() {
        System.out.println("Goodbye");
    }
}
```



NOT LEGAL.
Same
parameter lists
and same
names.

Cannot
distinguish
between the
two.

- **Is this legal?**

```
public class Test
{
    public void H()
    {
        System.out.println("Hello");
    }

    public void H(String m) {
        System.out.println("Goodbye");
    }
}
```

- **YES. *It is legal!!!***

```
public class Test
```

```
{
```

```
    public void H()
```

```
    {
```

```
        System.out.println("Hello");
```

```
    }
```

```
    public void H(String m) {
```

```
        System.out.println("Goodbye");
```

```
    }
```

```
}
```

LEGAL. Same name *but* different parameter list so OK.

Signatures are different!

- **Overloading**

- Same name ***but*** different parameter lists.
- Two methods can have the same name in the same scope as long as they have different parameter lists.
- If the parameter lists differ then the signatures will differ even if the method name is the same.

Overloading

- Is this legal?

```
public class Test {  
    public void H()  
    {  
        System.out.println("Hello");  
    }  
  
    public int H() {  
        System.out.println("Goodbye");  
        return 10;  
    }  
}
```

- **NO. *It is NOT legal!!!***

```
public class Test {  
    public void H()  
    {  
        System.out.println("Hello");  
    }  
  
    public int H() {  
        System.out.println("Goodbye");  
        return 10;  
    }  
}
```

NOT LEGAL.
Same name *and*
same parameter
list.

Return type is
NOT part of the
method
signature!

- ***How do we initialize a variable?***

- For **primitive** types it is easy:

```
int hourlyWorked = 35;
```

```
double hourlyRate = 35.50;
```

```
bool hourlyEmployee = true;
```

Initialization - REVIEW

- Reference types are trickier.
- A special method called a ***constructor*** is used to initialize an instance of an object.
- Constructors are called when you call new on the object being created.
- For example...

Initialization - REVIEW

```
public class Person {  
    private int m_Age;
```

```
    public Person()  
    {  
        m_Age = 10;  
    }  
}
```

```
Person p;
```

```
p = new Person(); // Calls constructor
```

Constructor - REVIEW

- Default constructor takes no parameters.
- You can also create constructors that take parameters.
- For example...

Constructor - REVIEW

```
public class Person {  
    private int m_Age;  
  
    public Person(int age)  
    {  
        m_Age = age;  
    }  
}
```

```
Person p;
```

```
p = new Person(20); // Pass value into constructor
```

Constructor - REVIEW

- The name of the constructor is the name of the class.
- Can you create more than one constructor for a class? **YES!!!**
- What must be different about each constructor?
- You can have as many constructors as you like as long as **ALL** the method signatures are unique.
- For example...

Overloading Constructor

```
public class Person {  
    private int m_Age;  
  
    public Person() // Zero parameters  
    {  
        m_Age = 10;  
    }  
  
    public Person(int age) // One parameter  
    {  
        m_Age = age;  
    }  
}
```

Overloading Constructor

- We will now move on to overriding...

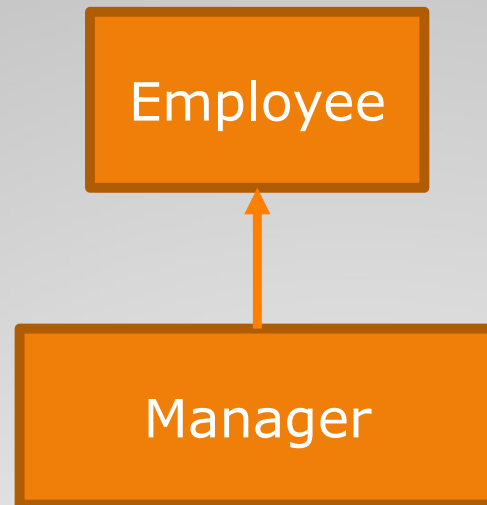
Overriding

- What is inheritance?
- A form of code reuse.
- Create a new class from an existing class.
- Use an existing class as a “base” for the new class.
- The new class adds on to the existing class.

Inheritance - REVIEW

- **Inheritance: “is-a” relationship**
- A derived class “is-a” type of the base class.
- **“A dog is an animal”**
- Base classes are more general than derived classes.

Inheritance - REVIEW



**Manager is derived
from Employee**

Inheritance - REVIEW

```
public class Employee
{
    protected int m_Id;

    public Employee(int newId)
    { m_Id = newId; }

    public int GetId()
    { return m_Id; }

    public void SetId(int newId)
    { m_Id = newId; }
}
```

Inheritance - REVIEW

```
public Manager extends Employee
```

**Manager is derived
from Employee**

```
{
```

```
    private String m_SecretaryName;
```

```
    public Manager(int newId, String newSec)
```

```
{
```

```
        super(newId); // Calls base class or superclass  
                        // constructor.
```

```
        m_SecretaryName = newSec;
```

```
}
```

```
// Assume other methods are declared here...
```

```
}
```

Inheritance - REVIEW

- Overload
 - ***Same method name*** BUT ***different signature***.
- Override
 - ***Same method name*** AND ***same signature***
 - Different implementation between classes in the inheritance hierarchy.
 - One implementation of the method is in the base class and the other is in the derived class.

Overloading Vs Overriding

Overload (different signature)	Override (same signature) Base Vs Derived
Employee::Show()	Employee::Show()
Employee::Show(int num)	HourlyEmployee::Show()

OVERLOAD

Each implementation of Show() has a DIFFERENT signature.

OVERRIDE

ALL implementations of Show() have the SAME signature. Each implementation of Show() differs between the base and derived class.

Overloading Vs Overriding

- **Polymorphism** means “**many forms**” in Greek.
- **IMPORTANT!!!**
Overloading AND overriding are two examples of ***polymorphism*** in programming.
- There were many different forms of the Show() method on the previous slide.

Polymorphism

- Write programs to process objects that share the same base class in a class hierarchy.
- Create a base class that other classes can derive from.
- The base class defines the common behavior that we care about.
- ***Put common behavior in the base class.***
- ***Program to the common behavior.***

Inheritance and Overriding

- In this example we will need to calculate and show an employee's weekly salary.
- We will do the following:
 - Define common behavior in the base class (Employee).
 - Override common behavior in derived classes.
- The common behavior in this example will be a method named ShowWeeklySalary.

Overriding Example

```
public class Employee {  
    protected double salary;  
  
    public Employee(double s) { salary = s; }  
    public double GetSalary() { return salary; }  
    public void SetSalary(double newSalary) { salary = newSalary; }  
  
    public void ShowWeeklySalary() {  
        double weeklySalary = salary / 52.0;  
  
        System.out.printf("Yearly Rate  = $%.2f\n", salary);  
        System.out.printf("Weekly Salary = $%.2f\n",weeklySalary);  
    }  
  
    }  
}
```

All employees presently at this company make a yearly salary. To calculate their weekly salary, we must divide by 52 (number of weeks in 1 year).

Overriding Example

```
public static void main(String[] args)
{
    Employee normalEmp = new Employee(52000);

    System.out.println("Weekly Salary Report");
    System.out.println("-----");

    normalEmp.ShowWeeklySalary();
}
```

**Show the
weekly salary**



Overriding Example

Hire Hourly Employees

- Now suppose we need to hire some workers who will get paid by the hour.
- Their weekly salary will get calculated differently.
- We can create a new class named HourlyEmployee that is derived from Employee.
- The logic to calculate an hourly employee's weekly salary will be in this new class.
- For example...

Overriding Example

```
public class HourlyEmployee extends Employee
```

```
{  
    public HourlyEmployee(double newSalary)  
    {  
        super(newSalary);  
    }  
}
```

@Override will cause a compile error to appear if the method being overridden does not exist on a base class (helps with spelling mistakes).

```
// OVERRIDE Employee::ShowWeeklySalary()
```

@Override is NOT required. Program will run fine without it.

```
@Override
```

```
public void ShowWeeklySalary()
```

```
{  
    double weeklySalary = salary * 40;
```

← Multiply salary (the hourly rate in this case) by 40 (total hours for week)

```
    System.out.printf("Hourly Rate   = $%.2f\n", salary);
```

```
    System.out.printf("Weekly Salary = $%.2f\n", weeklySalary);
```

```
    }  
}
```


Overriding Example

```
public static void main(String[] args)
```

```
{
```

```
    Employee normalEmp = new Employee(52000);
```

```
    Employee hourlyEmp = new HourlyEmployee(20);
```

 You can put an **HourlyEmployee** reference in an **Employee** variable because **HourlyEmployee** is derived from **Employee** (**HourlyEmployee** "is an" **Employee**)

```
    System.out.println("Weekly Salary Report");
```

```
    System.out.println("-----");
```

```
    normalEmp.ShowWeeklySalary();
```

```
    hourlyEmp.ShowWeeklySalary();
```

 Call **ShowWeeklySalary** on each **Employee** instance

```
}
```

Overriding Example

- How does the computer know which version of `ShowWeeklySalary()` to call?

Overriding

- Answer:

The underlying type determines which version of the method to call.

```
Employee normalEmp = new Employee(52000);  
Employee hourlyEmp = new HourlyEmployee(20);
```

```
// Calls Employee::ShowWeeklySalary()
```

```
normalEmp.ShowWeeklySalary();
```

```
// Calls HourlyEmployee::ShowWeeklySalary()
```

```
hourlyEmp.ShowWeeklySalary();
```

Overriding

- We can update the previous example's main code so that we store all employees in one array and process them in the same way.
- For example...

Overriding Example Revisited

```
public static void main(String[] args)
{
    Employee[] emps = new Employee[2];
    emps[0] = new Employee(52000);
    emps[1] = new HourlyEmployee(20);
```

← You can put an HourlyEmployee reference in the Employee array since HourlyEmployee is derived from Employee

```
System.out.println("Weekly Salary Report");
System.out.println("-----");
```

```
for (int i=0; i<emps.length; i++)
{
    emps[i].ShowWeeklySalary();
}
```

← The version of ShowWeeklySalary that gets called depends on the underlying type of the current array element.

Overriding Example Revisited

- We can further update the previous example's code.
- In the new version we will create a method that only has code related to Employee that can still process HourlyEmployees.
- For example...

Overriding Example Revisited

```
public static void main(String[] args) {  
    Employee[] emps = new Employee[2];  
    emps[0] = new Employee(52000);  
    emps[1] = new HourlyEmployee(20);
```

```
    report(emps);  
}
```

**Call the report method and
pass in the Employee array**



```
public static void report(Employee[] ea) {  
    System.out.println("Weekly Salary Report");  
    System.out.println("-----");
```

```
    for (int i=0; i<ea.length; i++)  
    {  
        ea[i].ShowWeeklySalary();  
    }  
}
```

**The report method is completely
based on Employee (no code
related to any other classes). Any
class that is derived from
Employee can be put into the array
and be processed by this method.**

Overriding Example Revisited

- What does the following code cause to happen?

```
class B {  
    private int salary;  
    public B() // Base constructor  
    { salary = 0; }  
}
```

Can you override the base
class constructor???

```
class D extends B {  
    @Override  
    public D() // Derived constructor  
    { }  
}
```

```
public class Driver {  
    public static void main(String[] args) {  
        D d = new D();  
    }  
}
```

Overriding

- What does the following code cause to happen?

```
class B {  
    private int salary;  
    public B() // Base constructor  
    { salary = 0; }  
}  
  
class D extends B {  
    @Override  
    public D() // Derived constructor  
    { }  
}  
  
public class Driver {  
    public static void main(String[] args) {  
        D d = new D();  
    }  
}
```

Can you override the base class constructor???

NO. Cannot override base class constructor!

1. To override you need to use the same name and parameter list.
2. Only one of the overridden methods in the inheritance hierarchy runs. We always need both constructors to run no matter what.

Overriding

- End of Slides

End of Slides