Inheritance (part 2)

November 13, 2006

ComS 207: Programming I (in Java)
Iowa State University, FALL 2006
Instructor: Alexander Stoytchev

Quick Review of Last Lecture

Parameter Passing (primitive types)

- **The act of passing an argument takes a copy of a value and stores it in a local variable acessible only to the method which is being called.**

```
{
  int num1=38;

Before:   num1  38

   myMethod(num1);

After:    num1  38

}
```

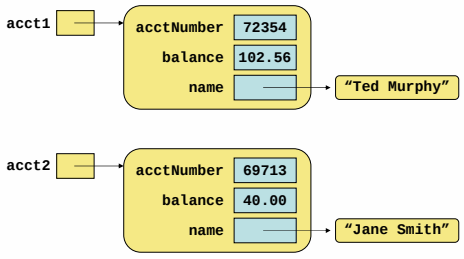
```
void myMethod(int num2)
{
  Before:   num2  38

   num2 =50;

  After:    num2  50

}
```


Objects and Reference Variables



Parameter Passing (objects)

- **Objects (in this case arrays) are also passed by value. In this case, however, the value is the address of the object pointed to by the reference variable.**

```
{
   int[] a={5, 7};
```

Before: a → [5, 7]

```
   myMethod(a);
```

After: a → [10, 7]

```
}
```

```
void myMethod(int[] b)
{
```

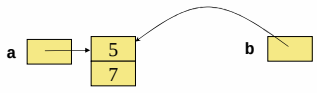
Before: b → [5, 7]

```
   b[0]+=5;
```

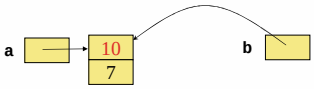
After: b → [10, 7]

```
}
```


In the previous example there is only one array and two references to it.



The array can be modified through either reference.



Method Overloading

- The compiler determines which method is being invoked by analyzing the parameters

```
float tryMe(int x)
{
   return x + .375;
}

float tryMe(int x, float y)
{
   return x*y;
}
```

Invocation

```
result = tryMe(25, 4.32)
```


Method Overloading

- The compiler determines which method is being invoked by analyzing the parameters

```
float tryMe(int x)
{
   return x + .375;
}
```

[signature 1] tryMe: int

```
float tryMe(int x, float y)
{
   return x*y;
}
```

[signature 2] tryMe: int, float

Method Overloading

- The `println` method is overloaded:

```
println (String s)
println (int i)
println (double d)
```

and so on...

- The following lines invoke different versions of the `println` method:

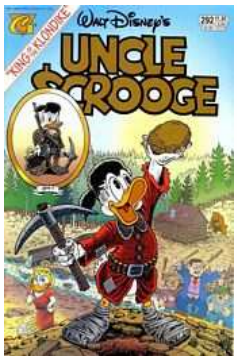
```
System.out.println ("The total is:");
System.out.println (total);
```


Inheritance

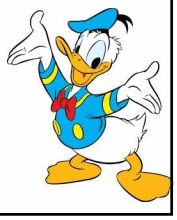
- Inheritance is a fundamental object-oriented design technique used to create and organize reusable classes
- Here is a quick analogy

Inheritance

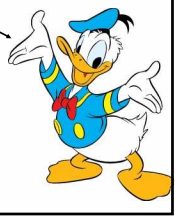
Inheritance



Inheritance



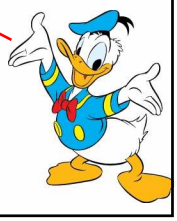
Inheritance



What can be inherited in Java?

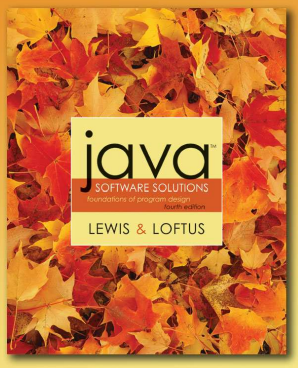
Inheritance

In class hierarchies the Inheritance arrow usually points up instead of down.



Chapter 8

Sections 8.1 & 8.2



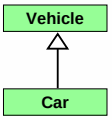
PEARSON Addison Wesley

Inheritance

- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class,* or *superclass*, or *base class*
- The derived class is called the *child class* or *subclass*
- As the name implies, the child inherits characteristics of the parent
- That is, the child class inherits the methods and data defined by the parent class

Inheritance

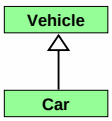
- Inheritance relationships are shown in a UML class diagram using a solid arrow with an unfilled triangular arrowhead pointing to the parent class



- Proper inheritance creates an *is-a* relationship, meaning the child *is a* more specific version of the parent

Inheritance

Class Hierarchy **Objects**




```
Vehicle v1 = new Vehicle();

Car c1 = new Car();
Car c2 = new Car();
Car c3 = new Car();
```


Inheritance

- A programmer can tailor a derived class as needed by adding new variables or methods, or by modifying the inherited ones
- *Software reuse* is a fundamental benefit of inheritance
- By using existing software components to create new ones, we capitalize on all the effort that went into the design, implementation, and testing of the existing software

Deriving Subclasses

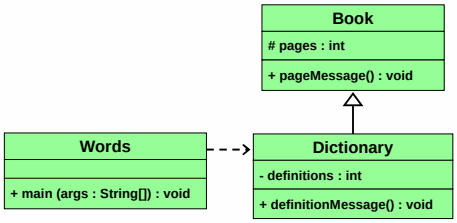
- In Java, we use the reserved word `extends` to establish an inheritance relationship

```
class Car extends Vehicle
{
   // class contents
}
```


Book & Dictionary Example

- See `Words.java` (page 440)
- See `Book.java` (page 441)
- See `Dictionary.java` (page 442)

Class Diagram for Words




```
public class Book
{
   protected int pages = 1500;

   public void setPages (int numPages)
   {
        pages = numPages;
   }

   public int getPages ()
   {
        return pages;
   }
  }
```



```
public class Dictionary extends Book
{
   private int definitions = 52500;


   public double computeRatio ()
   {
        return definitions/pages;
   }

   public void setDefinitions (int numDefinitions)
   {
        definitions = numDefinitions;
   }

   public int getDefinitions ()
   {
        return definitions;
   }
}
```



```
public class Dictionary extends Book
{
   private int definitions = 52500;









   public double computeRatio ()
   {  return definitions/pages; }


   public void setDefinitions (int numDefinitions)
   {     definitions = numDefinitions;  }

   public int getDefinitions ()
   {      return definitions; }
}
```



```
public class Dictionary extends Book
{
   private int definitions = 52500;
   protected int pages = 1500;

   public void setPages (int numPages)
   {      pages = numPages;  }

   public int getPages ()
   {     return pages;  }

   public double computeRatio ()
   {  return definitions/pages; }


   public void setDefinitions (int numDefinitions)
   {     definitions = numDefinitions;  }

   public int getDefinitions ()
   {      return definitions; }
}
```

INHERITED

The protected Modifier

- Visibility modifiers affect the way that class members can be used in a child class
- Variables and methods declared with private visibility cannot be referenced by name in a child class
- They can be referenced in the child class if they are declared with public visibility -- but public variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: `protected`

The protected Modifier

- The `protected` modifier allows a child class to reference a variable or method directly in the child class
- It provides more encapsulation than public visibility, but is not as tightly encapsulated as private visibility
- A protected variable is visible to any class in the same package as the parent class
- The details of all Java modifiers are discussed in Appendix E
- Protected variables and methods can be shown with a # symbol preceding them in UML diagrams

Appendix E

Modifier	Classes and interfaces	Methods and variables
default (no modifier)	Visible in its package.	Visible to any class in the same package as its class.
public	Visible anywhere.	Visible anywhere.
protected	N/A	Visible by any class in the same package as its class.
private	Visible to the enclosing class only	Not visible by any other class.

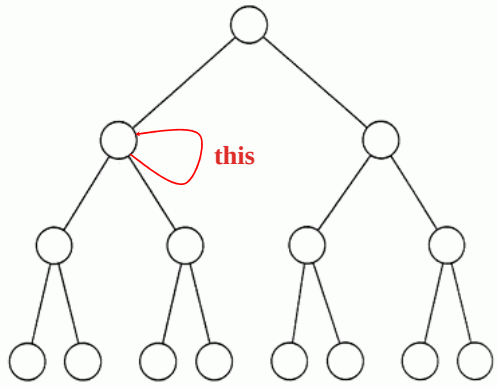
Modifier	Class	Interface	Method	Variable
abstract	The class may contain abstract methods. It cannot be instantiated.	All interfaces are inherently abstract. The modifier is optional.	No method body is defined. The method requires implementation when inherited.	N/A
final	The class cannot be used to drive new classes.	N/A	The method cannot be overridden.	The variable is a constant, whose value cannot be changed once initially set.
native	N/A	N/A	No method body is necessary since implementation is in another language.	N/A
static	N/A	N/A	Defines a class method. It does not require an instantiated object to be invoked. It cannot reference non-static methods or variables. It is implicitly final.	Defines a class variable. It does not require an instantiated object to be referenced. It is shared (common memory space) among all instances of the class.
synchronized	N/A	N/A	The execution of the method is mutually exclusive among all threads.	N/A
transient	N/A	N/A	N/A	The variable will not be serialized.
volatile	N/A	N/A	N/A	The variable is changed asynchronously. The compiler should not perform optimizations on it.

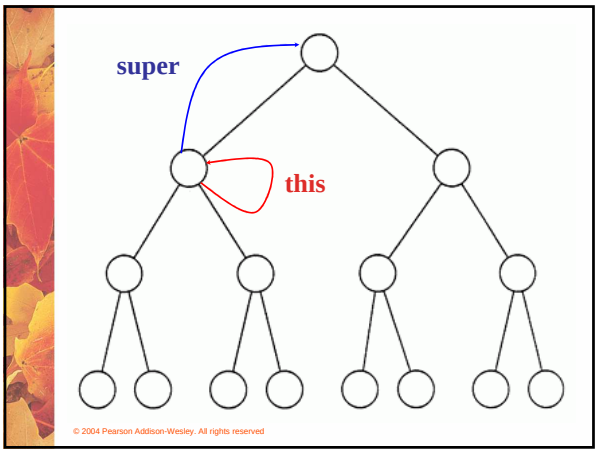
The super Reference

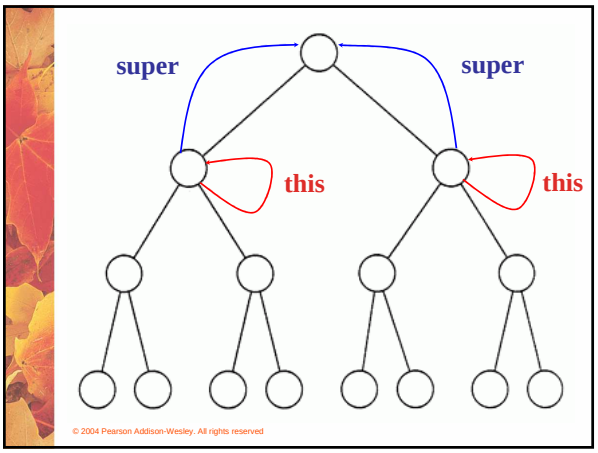
- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- The `super` reference can be used to refer to the parent class, and often is used to invoke the parent's constructor

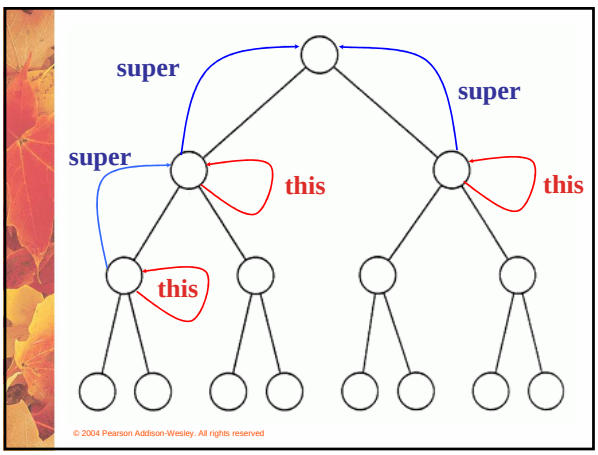
The super Reference

- A child's constructor is responsible for calling the parent's constructor
- The first line of a child's constructor should use the `super` reference to call the parent's constructor
- The `super` reference can also be used to reference other variables and methods defined in the parent's class





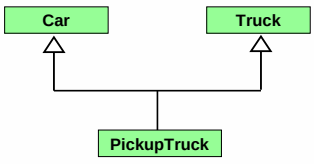




Modified Book Example

- See `Words2.java` (page 445)
- See `Book2.java` (page 446)
- See `Dictionary2.java` (page 447)

Multiple Inheritance

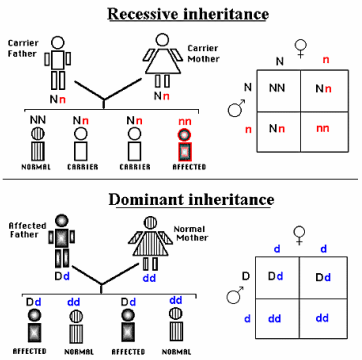


(not allowed in Java)

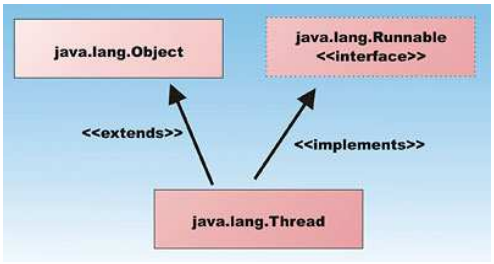
Multiple Inheritance

- Java supports *single inheritance*, meaning that a derived class can have only one parent class
- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents
- Collisions, such as the same variable name in two parents, have to be resolved
- Java does not support multiple inheritance
- In most cases, the use of interfaces gives us aspects of multiple inheritance without the overhead

Analogy



This example shows how multiple inheritance can be faked in java



[http://www.vsj.co.uk/pix/articleimages/may05/javathread3.jpg]

THE END