Java Inheritance

Inheritance

In this lesson of the Java tutorial, you will learn...

1. Understand the OOP concept of inheritance
2. Examine the role of inheritance in a program
3. Declare and use Java classes that extend existing classes
4. Understand the concept and role of polymorphism in Java

Inheritance

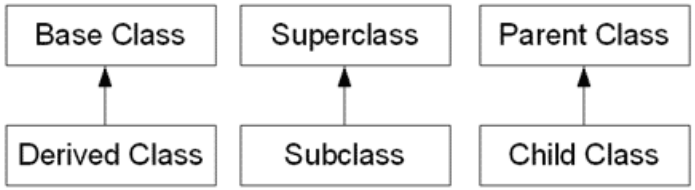
Inheritance creates a new class definition by building upon an existing definition (you *extend* the original class)

The new class can, in turn, can serve as the basis for another class definition

- all Java objects use inheritance
- every Java object can trace back up the inheritance tree to the generic class Object

The keyword extends is used to base a new class upon an existing class

Several pairs of terms are used to discuss class relationships (these are not keywords)



- note that traditionally the arrows point from the inheriting class to the base class, and the base class is drawn at the top - in the *Unified Modeling Language* (*UML*) the arrows point from a class to another class that it depends upon (and the derived class depends upon the base class for its inherited code)
- the parent class/child class terms are not recommended, since parent and child is more commonly used for ownership relationships (like a GUI window is a parent to the components placed in it)

A derived class instance may be used in any place a base class instance would work - as a variable, a return value, or parameter to a method

Inheritance is used for a number of reasons (some of the following overlap)

- to model real-world hierarchies
- to have a set of pluggable items with the same "look and feel," but different internal workings
- to allow customization of a basic set of features
- when a class has been distributed and enhancements would change the way existing methods work (breaking existing code using the class)
- to provide a "common point of maintenance"

When extending a class, you can add new properties and methods, and you can change the behavior of existing methods (which is called *overriding* the methods)

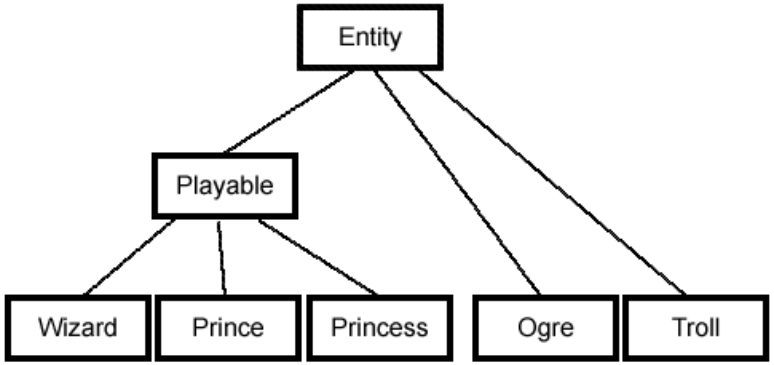
- you can declare a method with the same signature and write new code for it

- you can declare a property again, but this does not replace the original property - it *shadows* it (the original property exists, but any use of that name in this class and its descendants refers to the memory location of the newly declared element)

Inheritance Examples

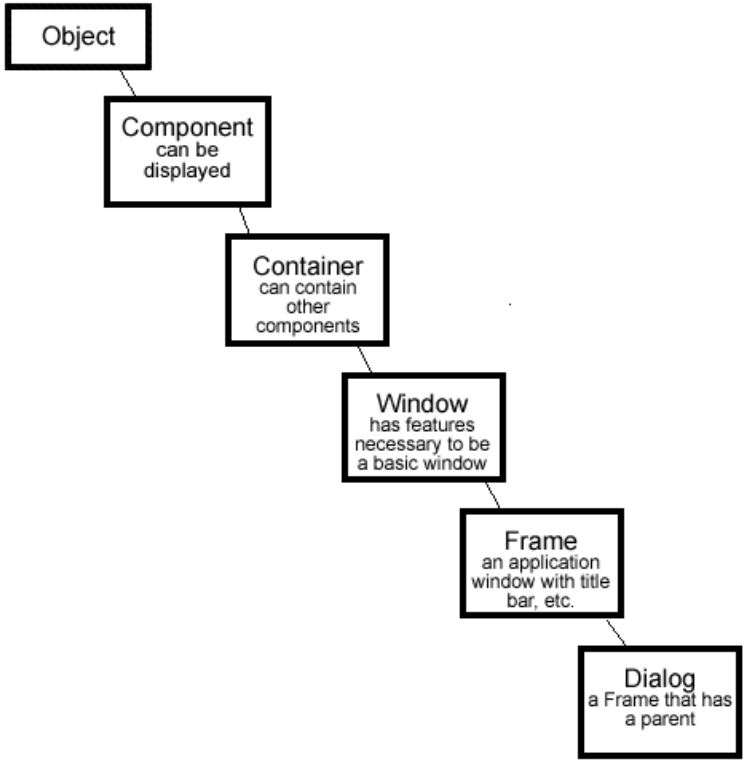
Say you were creating an arcade game, with a number of different types of beings that might appear - wizards, trolls, ogres, princesses (or princes), frogs, etc.

- all of these entities would have some things in common, such as a name, movement, ability to add/subtract from the player's energy - this could be coded in a base class Entity
- for entities that can be chosen and controlled by a player (as opposed to those that merely appear during the course of the game but can't be chosen as a character) a new class Playable could extend Entity by adding the control features
- then, each individual type of entity would have its own special characteristics - those that can be played would extend Playable, the rest would simply extend Entity
- you could then create an array that stored Entity objects, and fill it with randomly created objects of the specific classes
- for example, your code could generate a random number between 0 and 1; if it is between 0.0 and 0.2, create a Wizard, 0.2 - 0.4 a Prince, etc.



The Java API is a set of classes that make extensive use of inheritance

- one of the classes used in the GUI is Window - its family tree looks like:



Payroll with Inheritance

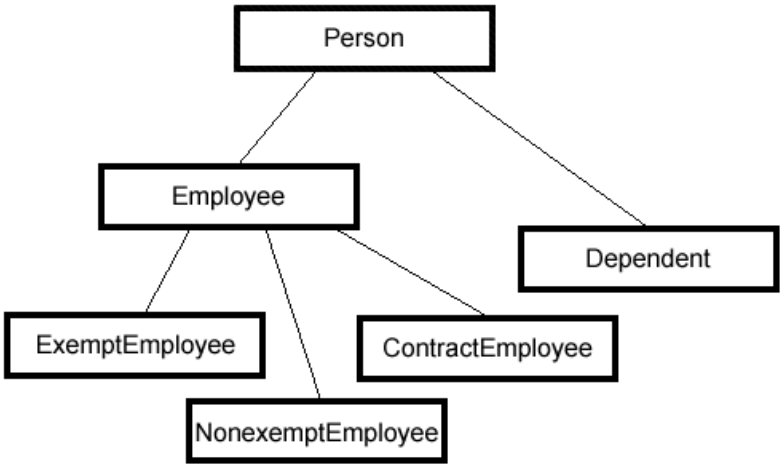
Our payroll program could make use of inheritance if we had different classes of employees: exempt employees, non-exempt employees, and contract employees

- they all share basic characteristics such as getting paid (albeit via different algorithms), withholding, having to accumulate year-to-date numbers for numerous categories
- but they have different handling regarding payment calculations, benefits, dependents, etc.
- exempt employees get a monthly salary, while nonexempt get a wage * hours, contract employees are handled similarly to nonexempt, but cannot have dependents

Also, we have already seen some duplication of effort in that our dependents store some of the same information as employees (first and last names)

- they use this information for the same purposes, so it might make sense to pull that common information into a base class

This would leave us with an inheritance scheme as follows:

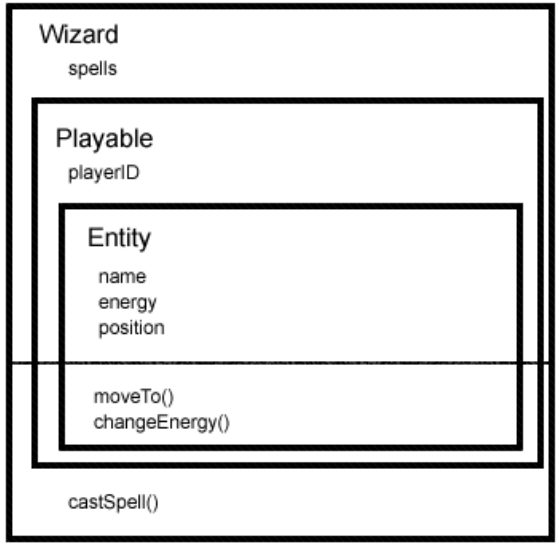


Note that a scheme with ContractEmployee extending NonexemptEmployee might also be a reasonable approach

Derived Class Objects

You can view a derived class object as having a complete base class object inside it

- lets assume that the Entity class defines the properties name, energy, and position, and methods moveTo() and changeEnergy()
- the Playable class adds a property playerID
- the Wizard class adds a spells property (an array of spells they can cast) and a castSpell() method



Any Wizard object contains all the elements inside its box, include those of the base classes

- so, for example, the complete set of properties in a Wizard object is:
  - name
  - energy
  - position
  - playerID
  - spells
- a Wizard reference to a Wizard object has access to any public elements from any class in the inheritance chain from Object to Wizard
- code inside the Wizard class has access to all elements of the base classes (except those defined as private in the base class - those are present, but not directly accessible)
- a more complete description of access levels is coming up in a few pages

Note: although it appears that a base class object is physically located inside the derived class instance, it is not actually implemented that way

Polymorphism

Inheritance and References

If a derived class extends a base class, it is not only considered an instance of the derived class, but an instance of the base class as well

- the compiler knows that all the features of the base class were inherited, so they are still there to work in the derived class (keeping in mind that they may have been changed)

This demonstrates what is known as an *IsA* relationship - a derived class object *Is A* base class instance as well

- it is an example of *polymorphism* - that one reference can store several different types of objects
- for example, in the arcade game example, for any character that is used in the game, an Entity reference variable could be used, so that at runtime, any subclass can be instantiated to store in that variable

```
Entity shrek = new Ogre();
```

```
Entity merlin = new Wizard();
```

- for the player's character, a Playable variable could be used

```
Playable charles = new Prince();
```

When this is done, however, the only elements immediately available through the reference are those know to exist; that is, those elements defined in the *reference type* object

- the compiler decides what to allow you to do with the variable based upon the type *declared for the variable*
- merlin.moveTo() would be legal, since that element is guaranteed to be there
- merlin.castSpell() would not be legal, since the definition of Entity does not include it, even though the actual object referenced by w does have that capability
- the following example gives a hint as to why this is the case:

```
Entity x;
if (Math.random() < 0.5) x = new Wizard();
else x = new Troll();
```

there is no way the compiler could determine what type of object would actually be created

- the variables names above, shrek, merlin, and charles, are probably not good choices: presumably we know shrek is an ogre, and always will be, so the type might as well be Ogre (unless, of course, he could transmogrify into something else during the game ...)

Dynamic Method Invocation

When a method is called through a reference, the JVM looks to the actual class of the instance to find the method. If it doesn't find it there, it backs up to the ancestor class (the class this class extended) and looks there (and if it doesn't find it there, it backs up again, potentially all the way to Object).

Sooner or later, it will find the method, since if it wasn't defined somewhere in the chain of inheritance, the compiler would not have allowed the class to compile.

In this manner, what you could consider the most advanced (or most derived) version of the method will run, even if you had a base class reference.

So, for our arcade game, an Entity reference could hold a Wizard, and when the move method is called, the Wizard version of move will run.

An interesting aspect of dynamic method invocation is that it occurs even if the method is called from base class code. If, for example:

- the Entity class move method called its own toString method
- the Ogre class didn't override move, but did override toString
- for an Ogre stored in an Entity variable, the move method was called

The Entity version of move would run, but its call to toString would invoke the toString method from Ogre!

Creating a Derived Class

The syntax for extending a base class to create a new class is:

Syntax

```
 [modifiers] class DerivedClassName extends
BaseClassName {
```

```
  (new property and method definitions go here)
}
```

- if you do not extend any class, Java assumes you are extending Object by default

Your new class can use the properties and methods contained in the original class (subject to the note coming up in a few pages about access keywords), add new data properties and methods, or replace properties or methods

A derived class object may be stored in a base class reference variable without any special treatment - the reverse is not true, but can be forced

Java doesn't allow *multiple inheritance*, where one class inherits from two or more classes

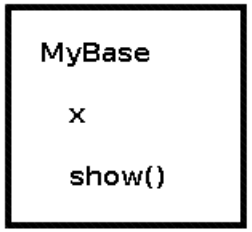
- it does have a concept called an *interface*, which defines a set of method names
- a class may implement an interface, defining those methods in addition to whatever other methods are in the class
- this allows for several otherwise unrelated classes to have the same set of method names available, and to be treated as the same type of object for that limited set of methods

Inheritance Example - A Derived Class

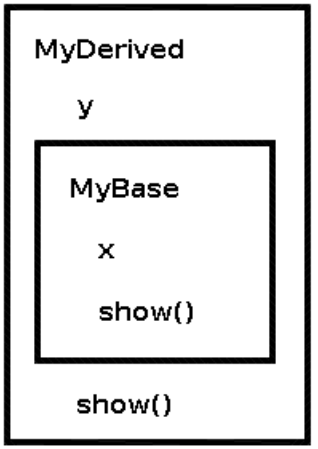
When a derived class is created, an object of the new class will in effect contain a complete object of the base class within it

The following maps out the relation between the derived class and the base class (note that the diagrams show the apparent memory allocation, in a real implementation the base class memory block is not inside the derived class block)

```
public class MyBase {
 public int x;
 public void show() {
  System.out.println("x =" + x);
}
```




```
class MyDerived extends MyBase {
 public int y;
 public void show() {
  System.out.println("x = " + x);
  System.out.println("y = " + y);
 }
}
```



Since everything in MyBase is public, code in the MyDerived class has free access to the x value from the MyBase object inside it, as well as y and show() from itself

- the show() method from MyBase is also available, but only within MyDerived class code (but some work is required to get it, since it is hidden by the show() method added with MyDerived) - code in other classes cannot invoke the MyBase version of

show() at all

Inheritance and Access

When inheritance is used to create a new (derived) class from an existing (base) class, everything in the base class is also in the derived class

- it may not be accessible, however - the access in the derived class depends on the access in the base class:

base class access	accessibility in derived class
public	public
protected	protected
private	inaccessible
unspecified (package access)	unspecified (package access)

Note that private elements become inaccessible to the derived class - this does not mean that they disappear, or that that there is no way to affect their values, just that they can't be referenced by name in code within the derived class

Also note that a class can extend a class from a different package

Inheritance and Constructors - the super Keyword

Since a derived class object contains the elements of a base class object, it is reasonable to want to use the base class constructor as part of the process of constructing a derived class object

- constructors are "not inherited"
- in a sense, this is a moot point, since they would have a different name in the new class, and can't be called by name under any circumstances, so, for example, when one calls new Integer(int i) they shouldn't expect a constructor named Object(int i) to run

Within a derived class constructor, however, you can use super(***parameterList***) to call a base class constructor

- it must be done as the first line of a constructor
- therefore, you can't use both this() and super() in the same constructor function
- if you do not explicitly invoke a form of super-constructor, then super() (the form that takes no parameters) will run
- and for the superclass, its constructor will either explicitly or implicitly run a constructor for its superclass
- so, when an instance is created, *one constructor will run at every level of the inheritance chain*, all the way from Object up to the current class

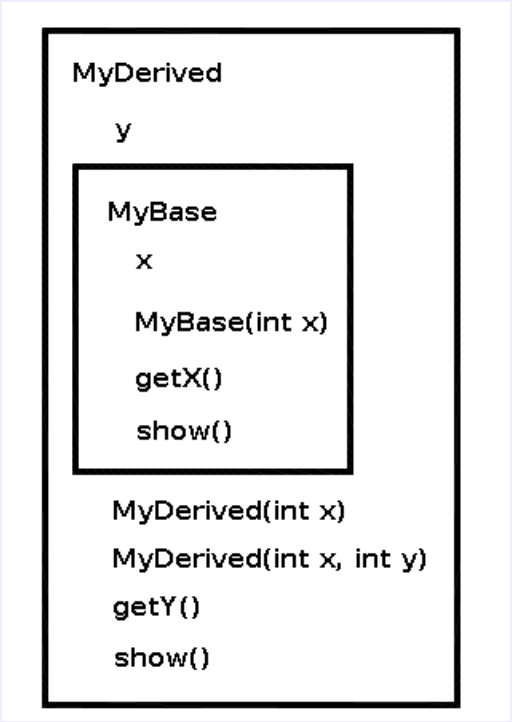
Code Sample: Java-Inheritance/Demos/Inheritance1.java

```
class MyBase {
 private int x;
  public MyBase(int x) {
  this.x = x;
 }
 public int getX() {
  return x;
 }
 public void show() {
  System.out.println("x=" + x);
 }
}
class MyDerived extends MyBase {
 private int y;
```

```
 public MyDerived(int x) {
  super(x);
 }
 public MyDerived(int x, int y) {
  super(x);
  this.y = y;
 }
 public int getY() {
  return y;
 }
 public void show() {
  System.out.println("x = " + getX());
  System.out.println("y = " + y);
 }
}
public class Inheritance1 {
 public static void main(String[] args) {
  MyBase b = new MyBase(2);
  b.show();
  MyDerived d = new MyDerived(3, 4);
  d.show(); }
}
```

Code Explanation

The diagram below shows the structure of our improved classes:



- a MyDerived object has two constructors available, as well as both the getX and getY methods and the show method
- both MyDerived constructors call the super constructor to handle storage of x
- the show method in the derived class overrides the base class version

- x from the base class is not available in the derived class, since it is private in MyBase, so the show method in MyDerived must call getX() to obtain the value

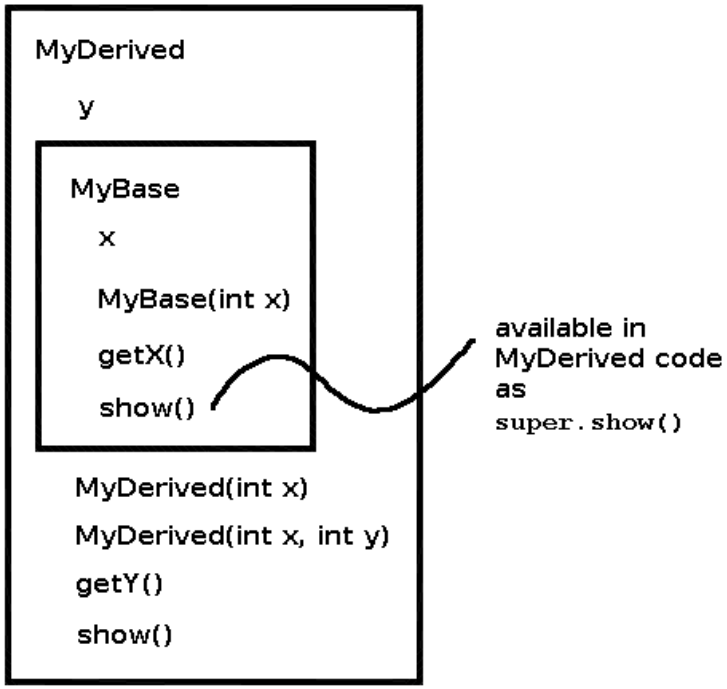
Derived Class Methods That Override Base Class Methods

As we saw before, you can create a method in the derived class with the same name as a base class method

- the new method *overrides* (and hides) the original method
- you can still call the base class method from within the derived class if necessary, by adding the super keyword and a dot in front of the method name
- the base class version of the method is not available to outside code
- you can view the super term as providing a reference to the base class object buried inside the derived class
- you cannot do super.super. to back up two levels
- you cannot change the return type when overriding a method, since this would make polymorphism impossible

Example: a revised MyDerived using super.show()

```
class MyDerived extends MyBase {
 int y;
 public MyDerived(int x) {
  super(x);
 }
 public MyDerived(int x, int y) {
  super(x);
  this.y = y;
 }
 public int getY() {
  return y;
 }
 public void show() {
  super.show();
  System.out.println("y = " + y);
 }
}
```



Inheritance and Default Base Class Constructors

One base class constructor will *always* run when instantiating a new derived class object

- if you do not explicitly call a base class constructor, the no-arguments base constructor will be automatically run, without the need to call it as super()
- but if you do explicitly call a base class constructor, the no-arguments base constructor will not be automatically run
- the no-arguments (or no-args for short) constructor is often called the default constructor, since it the one that will run by default (and also because you are given it by default if you write no constructors)

Code Sample: Java-Inheritance/Demos/Inheritance2.java

```
class Purple {
 protected int i = 0;
 public Purple() {
  System.out.println("Purple() running and i = " + i);
 }
 public Purple(int i) {
  this.i = i;
  System.out.println("Purple(i) running and i = " + i);
 }
}
class Violet extends Purple {
 Violet() {
  System.out.println("Violet() running and i = " + i);
 }
```

```
 Violet(int i) {
  System.out.println("Violet(i) running and i = " + i);
 }
}
public class Inheritance2 {
 public static void main(String[] args) {
  new Violet();
  new Violet(4);
 }
}
```

Code Explanation

Each constructor prints a message so that we can follow the flow of execution. Note that using new Violet() causes Purple() to run, and that new Violet(4) also causes Purple() to run.

If your base class has constructors, but no no-arguments constructor, then the derived class must call one of the existing constructors with super (***args***), since there will be no default constructor in the base class.

If the base class has a no-arguments constructor that is private, it will be there, but not be available, since private elements are hidden from the derived class. So, again, you must explicitly call an available form of base class constructor, rather than relying on the default

- try the above code with the Purple() constructor commented out or marked as private

The Instantiation Process at Runtime

In general, when an object is instantiated, an object is created for each level of the inheritance hierarchy. Ech level is completed before the next level is started, and the following takes place at each level:

1. the memory block is allocated
2. the entire block is zeroed out
3. explicit initializers run - which may involve executable code, for example: private double d = Math.random();
4. the constructor for this level runs
    - since the class code has already been loaded, and any more basic code has been completed, any methods in this class or inherited from superclasses are available to be called from the constructor
    - when you call a superconstructor, all you are really doing is selecting which form of superconstructor will run - timingwise, it was run before we got to this point

When the process has completed, the expression that created the instance evaluates to the address of the block for the last unit in the chain.

Example - Factoring Person Out of Employee and Dependent

Since dependents and personnel have some basic personal attributes in common, we will pull the first and last name properties into a base class representing a person

1. Create a new class, Person, cut the first and last name properties and the associated get and set methods out of Employee and paste them here (including getFullName)
2. Create a constructor for Person that sets the first and last names
3. Declare Employee to extend Person; change the constructor that accepts the first and last name properties to call a super-constructor to accomplish that task
4. Note that you will need to revise getPayInfo() - why?
5. Then, in a similar fashion, modify Dependent - in this case all you need to do is remove the name properties and associated methods, then modify the constructor to call the super-constructor to set the name elements
6. We can test the code using a simplified version of Payroll

Code Sample: Java-Inheritance/Demos/employees/Person.java

```
package employees;

public abstract class Person {
 private String firstName;
 private String lastName;

 public Person() {
 }

 public Person(String firstName, String lastName) {
  setFirstName(firstName);
  setLastName(lastName);
 }

 public String getFirstName() { return firstName; }

 public void setFirstName(String firstName) {
  this.firstName = firstName;
 }

 public String getLastName() { return lastName; }

 public void setLastName(String lastName) {
  this.lastName = lastName;
 }

 public String getFullName() {
  return firstName + " " + lastName;
 }

}
```

Code Explanation

This class includes the name properties and related set and get methods.

Code Sample: Java-Inheritance/Demos/employees/Dependent.java

```
package employees;

public class Dependent extends Person {

 private Employee dependentOf;

 public Dependent(Employee dependentOf, String firstName, String lastName) {
  super(firstName, lastName);
  this.dependentOf = dependentOf;
 }

 public Employee getDependentOf() {
  return dependentOf;
 }
}
```

Code Explanation

Since this class extends Person, the name-related elements are already present, so we remove them from this code.

Code Sample: Java-

Inheritance/Demos/employees/Employee.java

```
package employees;

public class Employee extends Person {
 private static int nextId = 1;
 private int id = nextId++;
 private int dept;
 private double payRate;
 private Dependent[] dependents = new Dependent[5];
 private int numDependents = 0;

 public Employee() {
 }
 public Employee(String firstName, String lastName) {
  super(firstName, lastName);
 }
 public Employee(String firstName,String lastName, int dept) {
  super(firstName, lastName);
  setDept(dept);
 }
 public Employee(String firstName, String lastName, double payRate) {
  super(firstName, lastName);
  setPayRate(payRate);
 }
 public Employee(String firstName, String lastName, int dept, double payRate) {
  this(firstName, lastName, dept);
  setPayRate(payRate);
 }

 public int getId() { return id; }

 public int getDept() { return dept; }

 public void setDept(int dept) {
  this.dept = dept;
 }

 public double getPayRate() { return payRate; }

 public void setPayRate(double payRate) {
  this.payRate = payRate;
 }

 public void addDependent(String fName, String lName) {
  if (numDependents < dependents.length) {
   dependents[numDependents++] = new Dependent(this, fName, lName);
  }
 }
 public String listDependents() {
  if (dependents == null) return "";
  StringBuffer temp = new StringBuffer();
  String newline = System.getProperty("line.separator");
  if (newline == null) newline = "\n";

  for (int i = 0; i < numDependents; i++) {
   temp.append(dependents[i].getFirstName());
   temp.append(" ");
   temp.append(dependents[i].getLastName());
   temp.append(newline);
  }
  return temp.toString();
```

```
 }

 public String getPayInfo() {
  return "Employee " + id + " dept " + dept + " " +
      getFullName() +
      " paid " + payRate;
 }
}
```

Code Explanation

The same changes that were made to Dependent can be made here. Note that since getPayInfo calls getFullName, which is now inherited and publicly accessible, that code did not need to change.

Code Sample: Java-Inheritance/Demos/Payroll.java

```
import employees.*;
import util.*;

public class Payroll {
 public static void main(String[] args) throws Exception {
  KeyboardReader kbr = new KeyboardReader();
  String fName = null;
  String lName = null;
  int dept = 0;
  double payRate = 0.0;
  double hours = 0.0;
  int numDeps = 0;
  String dfName = null;
  String dlName = null;

  Employee e = null;

  fName = kbr.getPromptedString("Enter first name: ");
  lName = kbr.getPromptedString("Enter last name: ");
  dept = kbr.getPromptedInt("Enter department: ");
  do {
   payRate = kbr.getPromptedFloat("Enter pay rate: ");
   if (payRate < 0.0) System.out.println("Pay rate must be >= 0");
  } while (payRate < 0.0);
  e = new Employee(fName, lName, dept, payRate);
  numDeps = kbr.getPromptedInt("How many dependents? ");
  for (int d = 0; d < numDeps; d++) {
   dfName = kbr.getPromptedString("Enter dependent first name: ");
   dlName = kbr.getPromptedString("Enter dependent last name: ");
   e.addDependent(dfName, dlName);
  }
  System.out.println(e.getPayInfo());
  System.out.println(e.listDependents());
 }
}
```

Code Explanation

No changes need to be made to Payroll to take advantage of the addition of the inheritance hierarchy that we added - the only changes we made were for the sake of brevity.

To revisit the sequence of events when instantiating a Dependent using the constructor that accepts the Employee and first and last names:

1. memory for an Object is allocated
2. any Object initializers would run
3. the Object() constructor would run
4. memory for a Person is allocated
5. it there were any Person initializers, they would run
6. the Person(String firstName, String lastName) constructor would run, because that was the version selected by the Dependent constructor we called
7. memory for a Dependent is allocated
8. it there were any Dependent initializers, they would run
9. the Dependent constructor we called would run

Exercise: Payroll-Inheritance01: Adding Types of Employees

Duration: 30 to 45 minutes.

We wish to improve our payroll system to take account of the three different types of employees we actually have: exempt, non-exempt, and contract employees. Rather than use some sort of identifying code as a property, OOP makes use of inheritance to handle this need, since at runtime a type can be programmatically identified.

Also, the solution code builds upon the Person base class from the preceding example. You can either copy the Person.java file into your working directory and edit Employee.java to match, or just copy the set of files from the Demos directory into a new working directory and use those.

1. Create three new classes: ExemptEmployee, NonexemptEmployee, and ContractEmployee that extend Employee (one possible approach is to copy the base class into new files and modify them)
2. In our company, exempt employees get a monthly salary, non-exempt an hourly rate that is multiplied by their hours, as do contract employees - revise the getPayInfo method to take this into account and also identify which type of employee it is (note that the payRate field will hold a monthly amount for exempt and an hourly amount for non-exempt and contractors, but these last two will need an additional field for their hours, plus methods to set and get the hours)
3. We need some way to handle the fact that contract employees do not have dependents - for now, just override listDependents to do nothing for a contract employee, and also override the addDependent method to do nothing for a ContractEmployee
    - we can see here a problem with the hard-coded instantiation of the dependents array in Employee
    - a better approach might be to leave it as null, and add a method to create the array, which could accept a size at that time (we won't bother to do this, since the collections classes offer a much better way that we will implement later)
4. Add constructors as you see fit
5. Have the main program create and display the information for instances of each type of employee (for now, comment out the code that uses the Employee array and just hard-code one of each type of employee)

Where is the solution?

Typecasting with Object References

Object references can be typecast only along a chain of inheritance

- if class MyDerived is derived from MyBase, then a reference to one can be typecast to the other

An *upcast* converts from a derived type to a base type, and will be done implicitly, because it is guaranteed that everything that could be used in the parent is also in the child class

```
Object o;
String s = new String("Hello");
```

```
o = s;
```

A *downcast* converts from a parent type to a derived type, and must be done explicitly

```
Object o = new String("Hello");
String t;
t = (String) o;
```

- even though o came from a String, the compiler only recognizes it as an Object, since that is the type of data the variable is declared to hold

As a memory aid, if you draw your inheritance diagrams with the parent class on top, then an upcast moves up the page, while a downcast moves down the page

More on Object Typecasts

The compiler will not check your downcasts, but the JVM will check at runtime to make sure that the cast is feasible

- a downcast could fail at runtime because you might call a method that is not there, so the JVM checks when performing the cast in order to fail as soon as possible, rather than possibly having some additional steps execute between casting and using a method

```
Object o = new Integer(7);
String t, u;

// compiler will allow this, but it will fail at runtime
t = (String) o;
u = t.toUpperCase();
```

- since t was not actually a String, this would fail with a runtime exception during the cast operation - at that point the runtime engine sees that it cannot make the conversion and fails

You can use a typecast in the flow of an operation, such as:

```
MyBase rv = new MyDerived(2, 3);
System.out.println("x=" + rv.getX() + " y=" + ((MyDerived) rv).getY() );
```

- ((MyBase) rv) casts rv to a MyDerived, for which the getY() method is run
- note the parentheses enclosing the inline typecast operation; this is because the dot operator is higher precedence than the typecast; without them we would be trying to run rv.getY() and typecast the result

Typecasting, Polymorphism, and Dynamic Method Invocation

The concept of *polymorphism* means "one thing - many forms"

In this case, the one thing is a base class variable; the many forms are the different types derived from that base class that can be stored in the variable

Storing a reference in a variable of a base type does not change the contents of the object, just the compiler's identification of its type - it still has its original methods and properties

- you must explicitly downcast the references back to their original class in order to access their unique properties and methods
- if you have upcast, to store a derived class object with a base class reference, the compiler will not recognize the existence of derived class methods that were not in the base class
- the collection classes, such as Vector, are defined to store Objects, so that anything you store in them loses its identity
  - you must downcast the reference back to the derived type in order to access those methods
  - the introduction of *generics* in Java 1.5 provides a solution to this annoyance (more on this in the Collections chapter)

During execution, using a base class reference to call to a method that has been overridden in the derived class will result in the derived class version being used - this is called *dynamic method invocation*

The following example prints the same way twice, even though two different types of variable are used to reference the object:

Code Sample: Java-Inheritance/Demos/Inheritance3.java

```
public class Inheritance3 {
 public static void main(String[] args) {
  MyDerived mD = new MyDerived(2, 3);
  MyBase mB = mD;
  mB.show();
  mD.show();
 }
}
```

More on Overriding

Changing access levels on overridden methods

You can change the access level of a method when you override it, but only to make it more accessible

- you can't restrict access any more than it was in the base class
- so, for example, you could take a method that was protected in the base class and make it public
- for example, if a method was public in the base class, the derived class may not override it with a method that has protected, private or package access

This avoids a logical inconsistency:

- since a base class variable can reference a derived class object, the compiler will allow it to access something that was public in the base class
- if the derived class object actually referenced had changed the access level to private, then the element ought to be unavailable
- this logic could be applied to any restriction in access level, not just public to private

As a more specific example of why this is the case, imagine that ExemptEmployee overrode public String getPayInfo() with private String getPayInfo()

The compiler would allow

```
Employee e = new ExemptEmployee();

// getPayInfo was public in Employee, so compiler should allow this
e.getPayInfo();
```

- because Employee, the type on the variable e, says that getPayInfo is public
- but, now at runtime, it shouldn't be accessible, since it is supposed to be private in ExemptEmployee

Redefining properties

A property in a derived class may be redefined, with a different type and/or more restrictive access - when you do this you are creating a second property that hides the first; this is called *shadowing* instead of overriding

- a new property is created that hides the existence of the original property
- since it actually a new property being created, the access level and even data type may be whatever you want - they do not have to be the same as the original property
- I don't know of any good reason to do this, but it is possible
- a strange thing happens when you use a base class reference to such a class where the property was accessible (for example, public)
  - the base class reference sees the base class version of the property!

Object Typecasting Example

Say our game program needed to store references to Entity objects

- the exact type of Entity would be unknown at compile time
- but during execution we would like to instantiate different types of Entity at random

Perhaps our code for Entity includes a method move() that moves it to a new location. Many of the entities move the same way, but perhaps some can fly, etc. We could write a generally useful form of move in the Entity class, but then override it as necessary in some of the classes derived from Entity.

Code Sample: Java-Inheritance/Demos/EntityTest.java

```
class Entity {
 private String name;
 public Entity(String name) { this.name = name; }
 public String getName() { return name; }
 public void move() {
  System.out.println("I am " + name + ". Here I go!");
 }
}
class Playable extends Entity {
 public Playable(String name) { super(name); }
 public void move() {
  System.out.println("I am " + getName() + ". Here we go!");
 }
}
class Ogre extends Entity {
 public Ogre(String name) { super(name); }
}
class Troll extends Entity {
 public Troll(String name) { super(name); }
}
class Princess extends Playable {
 public Princess(String name) { super(name); }
 public void move() {
  System.out.println("I am " + getName() + ". Watch as I and my court move!");
 }
}
class Wizard extends Playable {
 public Wizard(String name) { super(name); }
 public void move() {
  System.out.println("I am " + getName() + ". Watch me translocate!");
 }
}
public class EntityTest {
 public static void main(String[] args) {
  String[] names = { "Glogg", "Blort", "Gruff",
```

```
            "Gwendolyne", "Snow White", "Diana",
            "Merlin", "Houdini", "Charles", "George" };
  for (int i = 0; i < 10; i++) {
   int r = (int) (Math.random() * 4);
   Entity e = null;
   switch (r) {
    case 0: e = new Ogre(names[i]); break;
    case 1: e = new Troll(names[i]); break;
    case 2: e = new Wizard(names[i]); break;
    case 3: e = new Princess(names[i]); break;
   }
   e.move();
  }
 }
}
```

The compiler allows the calls to the move() method because it is guaranteed to be present in any of the subclasses - since it was created in the base class

- if not overridden in the derived class, then the base class version will be used
- if the method is overridden by the derived class, then the derived class version will be used

At runtime, the JVM searches for the method implementation, starting at the actual class of the instance, and moving up the inheritance hierarchy until it finds where the method was implemented, so the most derived version will run.

Checking an Object's Type: Using instanceof

Given that you can have base class references to several different derived class types, you will eventually come to a situation where you need to determine exactly which derived class is referenced - you may not know it at the time the program is compiled

- in the above example, perhaps wizards, ogres, trolls, etc. have their own special methods
- how would you know which method to call if you had an Entity reference that could hold any subclass at any time?

The instanceof operator is used in comparisons - it gives a boolean answer when used to compare an object reference with a class name

Syntax

```
referenceVariable instanceof ObjectType
```

- it will yield true if the object is an instance of that class
- it will also give true if the object is a derived class of the one tested
- if the test yields true, then you can safely typecast to call a derived class method (you still need to typecast to call the method - the compiler doesn't care that you performed the test)

For example:

```
if (e[i] instanceof Wizard) ((Wizard) e[i]).castSpell();
```

There is another method of testing that is exact:

- every object has a getClass() method that returns a Class object that uniquely identifies each class
- every class has a class property that provides the same Class object as above; this object is the class as loaded into the JVM (technically, class isn't a property, but syntax-wise it is treated somewhat like a static property)
- with this method, a derived class object's class will compare as not equal to the base class

```
if (e[i].getClass().equals(Wizard.class)) ((Wizard) e[i]).castSpell();
```

- it is rare that you would need this type of test

Typecasting with Arrays of Objects

Unlike with arrays of primitives, it is possible to typecast an array of one type of object to an array of another type of object, if the types are compatible. The following example converts an array of strings to an array of objects and back again:

Code Sample: Java-Inheritance/Demos/ObjectArrayTypecast.java

```
public class ObjectArrayTypecast {

 public static void main(String[] args) {

  Object[] objects;
  String[] strings = { "A", "B", "C", "D" };

  objects = strings;

  for (Object o : objects) System.out.println(o);

  strings = null;
  strings = (String[]) objects;
  for (String s : strings) System.out.println(s);

 }
}
```

Code Explanation

Both upcasts and downcasts can be made. Because the arrays store references, the physical attributes, like the size, of each element remains the same regardless of what type of object the element references.

Exercise: Payroll-Inheritance02: Using the Employee Subclasses

Duration: 45 to 60 minutes.

1. In the main method of Payroll, reinstate the array of Employee objects
2. Create instances of the non-exempt employees, exempt employees, and contract employees to fill the array
    - you can ask which type of employee using "E", "N", and "C", then use if … else or a switch to control the input and the instantiation of the specific type of employee (if using if … else, Character.toUpperCase(char c) may come in handy)
3. Create a report that lists all employees, grouped by type, by looping through the array three times
    - the first time, show all exempt employees and their pay information
        - also list their dependents

- the second time. print only the non-exempt employees and their pay information
  - also list their dependents
- the third time. print contract employees and their pay information
- since you'll be writing the same loop multiple times, you could try both indexed loops and for-each loops (the solution uses for-each loops)

Where is the solution?

Other Inheritance-Related Keywords

abstract

States that the item cannot be realized in the current class, but can be if the class is extended

Syntax

```
abstract [other modifiers] dataType name ...
```

- for a class, it states that the class can't be instantiated (it serves merely as a base for inheritance)
- for a method, it states that the method is not implemented at this level
- the abstract keyword cannot be used in conjunction with final

abstract Classes

Used when a class is used as a common point of maintenance for subsequent classes, but either structurally doesn't contain enough to be instantiated, or conceptually doesn't exist as a real physical entity

```
public abstract class XYZ { ... }
```

- we will make the Employee class abstract in the next exercise: while the concept of an employee exists, nobody in our payroll system would ever be just an employee , they would be exempt, non-exempt, or contract employees
- while you cannot instantiate an object from an abstract class, you can still create a reference variable whose type is that class

abstract Methods

The method cannot be used in the current class, but only in a inheriting class that overrides the method with code

```
public abstract String getPayInfo();
```

- the method is not given a body, just a semicolon after the parentheses
- if a class has an abstract method, then the class must also be abstract
- you can extend a class with an abstract method without overriding the method with a concrete implementation, but then the class must be marked as abstract

final

Used to mark something that cannot be changed

Syntax

```
final [other modifiers] dataType name ...
```

final Classes

The class cannot be extended

```
public final class XYZ { ... }
```

final Methods

The method cannot be overridden when the class is extended

```
public final void display() { ... }
```

final Properties

Marks the property as a constant

```
public static final int MAX = 100;
```

- a final value can be initialized in a constructor or initializer:

```
public final double randConstant = Math.random();
```

or

```
public final double randConstant;
```

- then, in a constructor:

```
randConstant = Math.random();
```

- note: String and the wrapper classes use this in two ways :
  - the class is final, so it cannot be extended
  - the internal property storing the data is final as well, but set by the constructor (this makes the instance *immutable* - the

contents cannot be changed once set)
- in some cases, a declaration of final enables the compiler to optimize methods, since it doesn't have to leave any "hooks" in for potential future inheritance

Note that final and abstract cannot be used for the same element, since they have opposite effects.

Exercise: Payroll-Inheritance03: Making our base classes abstract

Duration: 5 to 5 minutes.

1. Mark the Person and Employee classes as abstract
2. Make the getPayInfo() method abstract in Employee

Where is the solution?

Methods Inherited from Object

There are a number of useful methods defined for Object

Some are useful as is, such as:

Class getClass() - returns a Class object (a representation of the class that can be used for comparisons or for retrieving information about the class)

Others are useful when overridden with code specific to the new class:

Object clone() - creates a new object that is a copy of the original object

- this method must be overridden, otherwise an exception will occur (the Object version of clone throws a CloneNotSupportedException)
- the issue is whether to perform a *shallow copy* or a *deep copy* - a shallow copy merely copies the same reference addresses, so that both the original object and the new object point to the same internal objects; a deep copy makes copies of all the internal objects (and then what if the internal objects contained references to objects …)

boolean equals(Object) - does a comparison between this object and another

- if you don't override this method, you get the same result as if you used == (that is, the two references must point to the same object to compare as equal - two different objects with the same properties would compare as unequal) - that is how the method is written in the Object class
- you would override this method with whatever you need to perform a comparison

int hashCode() - returns an integer value used by collection objects that store elements using a *hashtable*

- elements that compare as the same using the equals(Object) method should have the same hashcode

String toString() - converts this object to a string representation

- this method is called by a some elements in the Java API when the object is used in a situation that requires a String, for example, when you concatenate the object with an existing String, or send the object to System.out.println()
- note that the call to toString is not made automatically as a typecast to a String - the only behavior built into the syntax of Java is string concatenation with the + sign (so one of the operands must already be a String); the code in println is explicitly written to call toString
- if you don't override this method, you will get a strange string including the full class name and the *hashcode* for the object

void finalize() - called by the JVM when the object is garbage-collected

- this method might never be called (the program may end without the object being collected)

There are also a several methods (wait, notify, and notifyAll) related to locking and unlocking an object in multithreaded situations

Code Sample: Java-Inheritance/Demos/ObjectMethods.java

```
public class ObjectMethods {

  int id;
  String name;
  int age;
  ObjectMethods(int id, String name, int age) {
    this.id = id;
    this.name = name;
    this.age = age;
  }

  public boolean equals(Object x) {
    if (x == this) return true;
    else if (x instanceof ObjectMethods) {
      ObjectMethods omx = (ObjectMethods) x;
      return id == omx.id && name.equals(omx.name) && age == omx.age;
    }
    else return false;
  }

  public int hashCode() {
    return id + age * 1000;
  }

  public String toString() {
    return id + " " + name + " is " + age + " years old";
  }

  public Object clone() {
    return new ObjectMethods(id, name, age);
  }

  public static void main(String[] args) {
    ObjectMethods om1 = new ObjectMethods (1, "John", 6);
    ObjectMethods om2 = new ObjectMethods (1, "John", 6);
    ObjectMethods om3 = new ObjectMethods (2, "Jane", 5);
    ObjectMethods om4 = (ObjectMethods)om3.clone();
    System.out.println("Printing an object: " + om1);
    if (om1.equals(om2))
      System.out.println("om1 equals(om2)");
    if (om1.equals(om3))
      System.out.println("om1 equals(om3)");
    if (om1.equals("John"))
      System.out.println("om1 equals(\"John\")");
    if (om3.equals(om4))
      System.out.println("om3 equals(om4) which was cloned from om3");
    System.out.println("object class is: " + om1.getClass());
  }

}
```

Code Explanation

The clone method returns Object rather than ObjectMethods, since that is how it was declared in the Object class, and you can't change the return type when overriding - thus the typecast on the returned value.

Similarly, the parameter to equals is Object, rather than ObjectMethods. This is not required by Java's syntax rules, but rather a convention that enables other classes to work with this class. For example, the Collections API classes use the equals method to determine if an object is already in a set. If we wrote the method as equals(ObjectMethods om) instead of equals(Object o), the collections classes would call equals(Object o) as inherited from Object, which would test for identity using an == test.

The hashCode method was written out of a sense of duty - Sun specifies that the behavior of hashCode "should be consistent with equals", meaning that if two items compare as equal, then they should have the same hash code - this will be revisited in the section on Collections.

Inheritance Conclusion

In this lesson of the Java tutorial you have learned:

- The role of inheritance in a Java program
- How to declare and use classes that extend existing classes
- About methods inherited from Object