Inheritance & OOP (Object-Oriented Programming)

Chap. 11

(§11.1-11.4: Three good examples:

1. Simulation of aviary
2. Geological classification
3. Payroll)

Inheritance, OOD, and OOP

A major objective of OOP:

Write reusable code

(to avoid re-inventing the wheel).

Ways to do this in Java:

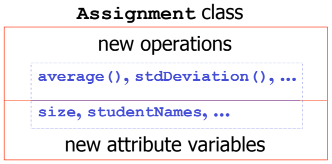
- Encapsulate code within *methods*
- Build *classes*
- Store classes in *packages*
- An additional approach that distinguishes OOP from the others: ***inheritance***

Example: Suppose we completed and tested our **`Assignment`** class of Ch. 10 & 11 to include various other statistics, grading-on-the-curve (ala Lab 9), etc.

Now suppose that sometime later we find that we need some other operations that aren't provided in this class — e.g., sort the list of names and scores, plot a histogram, etc.

How should we proceed? There are several alternatives:

#1. Add new attribute variables and methods to the **`Assignment`** class



`Assignment` class

new operations

`average(), stdDeviation(), ...`

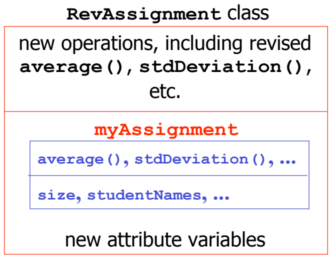
`size, studentNames, ...`

new attribute variables

But . . . this can easily mess up a tested, operational class, creating problems for other client programs (and isn't even possible if we were modifying a class provided in Java).

#2: A wrapper approach: Build a new **RevAssignment** class that contains an **Assignment** object as an attribute variable.

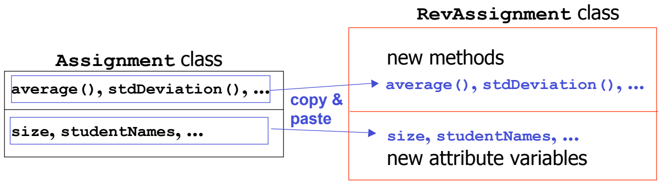
Composition


RevAssignment class
new operations, including revised
average(), stdDeviation(),
etc.
myAssignment
average(), stdDeviation(), …
size, studentNames, …
new attribute variables

Better, but:

A **RevAssignment** *is not an* **Assignment**;
it *has an* **Assignment**.

#3: Copy-&-paste approach: Build a new class, copying and pasting the attribute variables and methods of **Assignment** into **RevAssignment**.


RevAssignment class
Assignment class
average(), stdDeviation(), …
size, studentNames, …
copy & paste
new methods
average(), stdDeviation(), …
size, studentNames, …
new attribute variables

Almost right, but:

These are separate independent classes. Modifying **Assignment**(e.g., changing from using arrays for the names and scores to a different container) doesn't automatically update a **RevAssignment**.

#4: Object-oriented approach:

Define **RevAssignment** as an extension of **Assignment** so that it inherits the characteristics (attribute variables, constants, and methods) of **Assignment**, thus *reusing these variables, constants, and methods of that class*.

RevAssignment is called a subclass
(or child class or derived class)
of **Assignment**,
which is called a superclass
(or parent class or base class).

This is the best approach:

i. A child class inherits all the attributes and operations of its parent class; we need not reinvent the wheel!

ii. Modifying a parent class automatically updates every child class.

iii. Mistakes made in building **a child** class will be local to it; the original parent class remains unchanged so that any client programs using this class are not affected.

OOD (Object-Oriented Design)

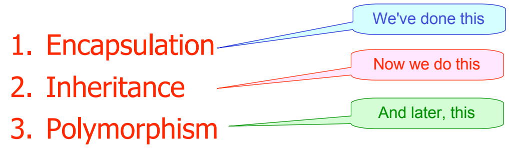
Object-oriented design (OOD) is to engineer one's software as follows:

1. Identify the objects in the problem
2. Look for *commonality* in those objects
3. Define superclasses containing that commonality
4. Define *subclasses* that inherit from the superclasses

These last two steps are the most difficult aspects of OOD.

OOP

Object-oriented programming (OOP) was first used to describe the programming environment for **Smalltalk**, the earliest true object-oriented programming language. OOP languages have three important properties:



1. Encapsulation
2. Inheritance
3. Polymorphism

Class Heirarchies — a Start

Problem: Model various kinds of licenses.

Old Approach: Build separate classes for each license from scratch

OOD: *What attributes and operations do all licenses have in common?*

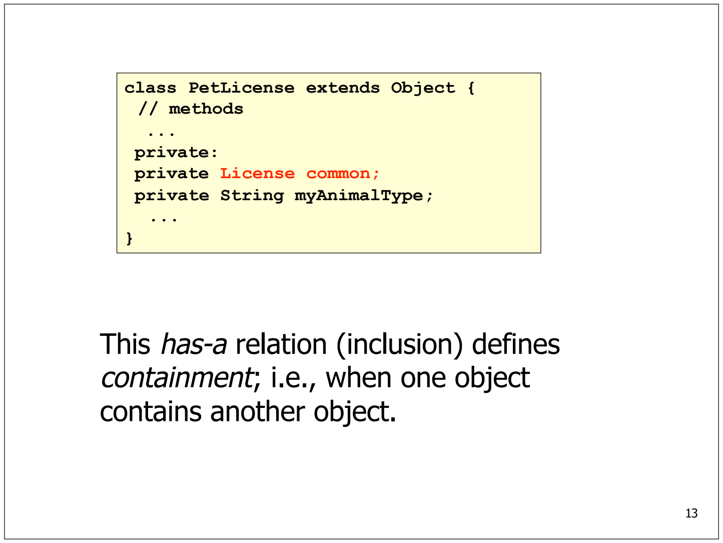
Then store these common attributes and operations in a general (base) class **`License`:**

```
class License extends Object
{
  // public methods:
  // constructor, accessors, output, ...

  private String myName;
  private int myAge;
  private int myIdNumber;
}
```

Now, to get these common attributes and operations into classes for each of the various kinds of licenses, we could give each of them an instance variable of type **`License`** and then add other instance variables that are specific to that type of license:

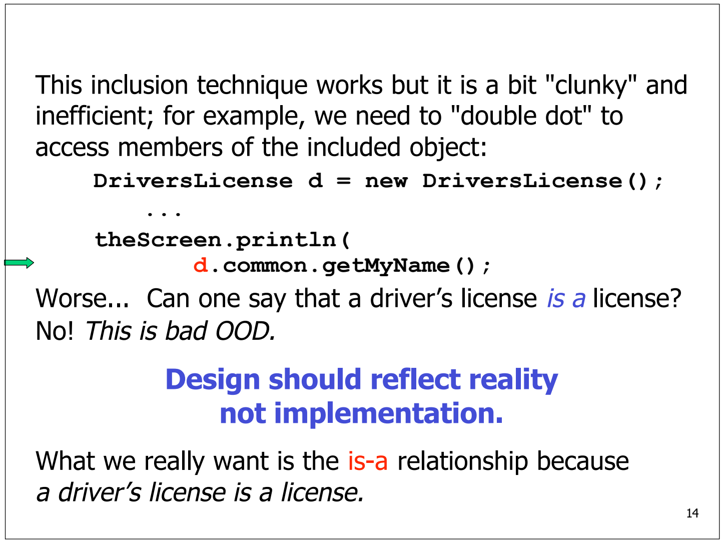
```
class DriversLicense extends Object {
 // methods
  ...
 private License common;
 private int myVehicleType;
 private String myRestrictionsCode;
  ...
}
```

```
class HuntingLicense extends Object {
{
 // methods
  ...
 private License common;
 private String myPrey;
  ...
}
```



```
class PetLicense extends Object {
  // methods
   ...
 private:
 private License common;
 private String myAnimalType;
   ...
}
```

This *has-a* relation (inclusion) defines *containment*; i.e., when one object contains another object.



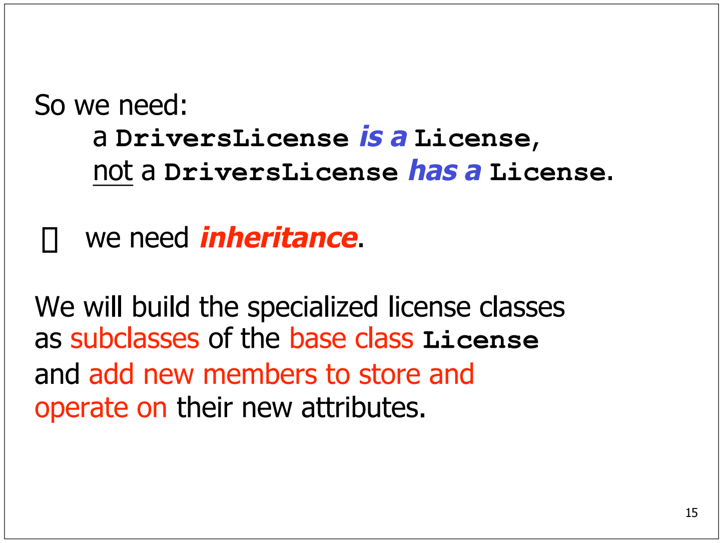
This inclusion technique works but it is a bit "clunky" and inefficient; for example, we need to "double dot" to access members of the included object:

```
    DriversLicense d = new DriversLicense();
        ...
    theScreen.println(
           d.common.getMyName();
```

Worse… Can one say that a driver’s license *is a* license? No! *This is bad OOD.*

Design should reflect reality
not implementation.

What we really want is the is-a relationship because *a driver’s license is a license.*

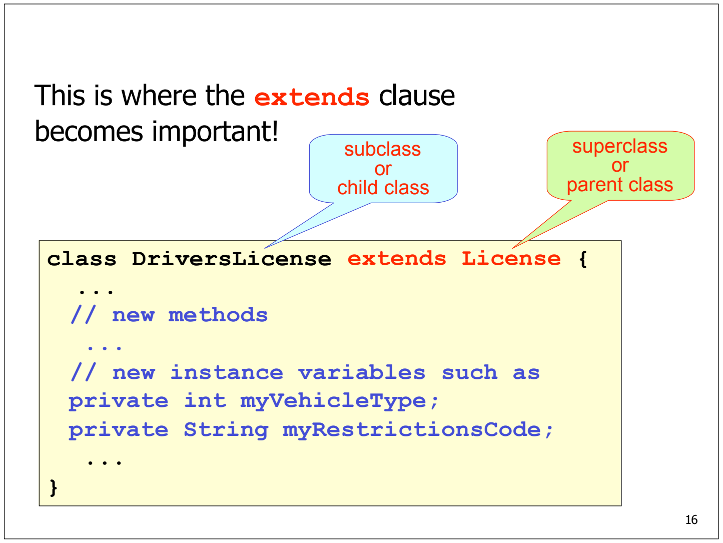


So we need:

a **DriversLicense** ***is a*** **License**,
not a **DriversLicense** ***has a*** **License**.

□ we need ***inheritance***.

We will build the specialized license classes as subclasses of the base class **License** and add new members to store and operate on their new attributes.



This is where the **extends** clause becomes important!

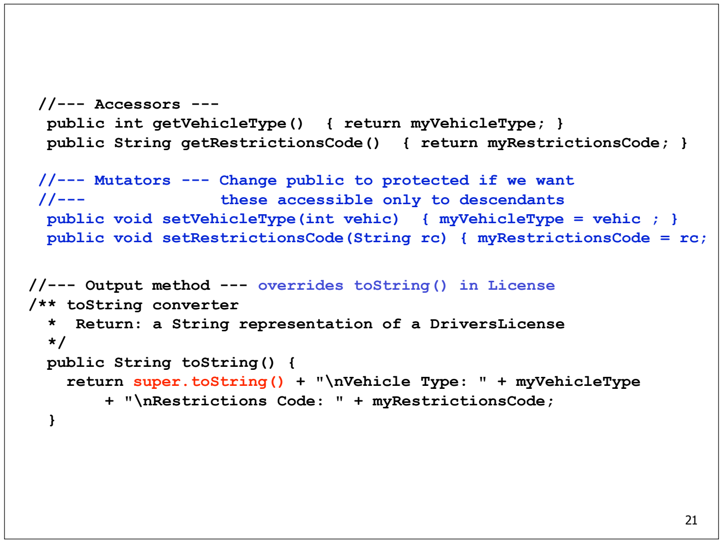
```
class DriversLicense extends License {
   ...
  // new methods
    ...
  // new instance variables such as
  private int myVehicleType;
  private String myRestrictionsCode;
    ...
}
```

```java
/** License.java provides a class to model licenses in general.
 */
class License extends Object {
 //--- Constructors ---
 /** Default constructor
  *  Postcondition: A License object is constructed with
  *                 myName == "", myAge == 0, myIdNumber == 0,
  */
  public License() {
    myName = "";
    myAge = 0;
    myIdNumber = 0;
  }

 /** Explicit-value constructor
  *  Receive:       String name, int age, int number
  *  Postcondition: A License object is constructed with
  *                 myName == name, myAge == age, myIdNumber == number.
  */
  public License(String name, int age, int number) {
    myName = name;
    if (age >= 0)  myAge = age;
    else           System.err.println("Invalid age value");
    if (number >= 0)  myIdNumber = number;
    else              System.err.println("Invalid id number");
  }
```

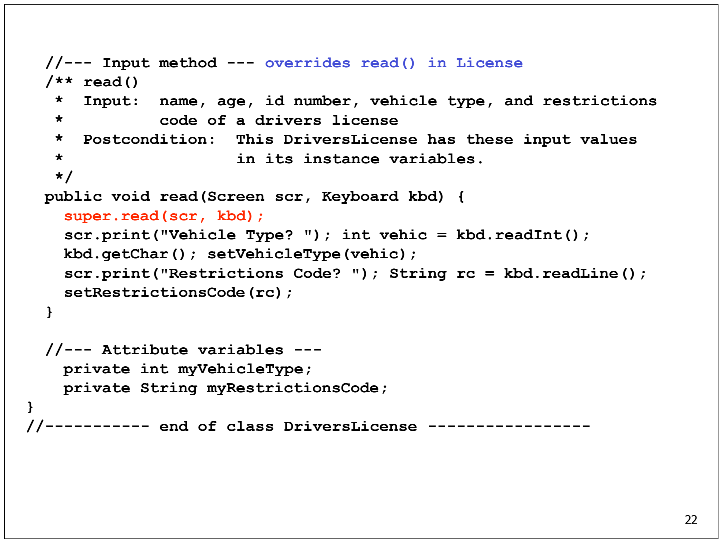
```java
 //--- Accessors ---
  public String getName()  { return myName; }
  public int getAge()      { return myAge; }
  public int getIdNumber() { return myIdNumber; }

 //--- Mutators --- Change public to protected if we want
 //---                 these accessible only to descendants
  public void setName(String name)  { myName = name; }
  public void setAge(int age) {
    if (age >= 0)  myAge = age;
    else           System.err.println("Invalid age value");
  }
  public void setIdNumber(int number) {
    if (number >= 0)  myIdNumber = number;
    else              System.err.println("Invalid id number");
  }

 //--- Output method ---
 /** toString converter
  *  Return: a String representation of a License
  */
  public String toString() {
    return myName  + "\n" + myAge + " years old"
                   + "\nID: " + myIdNumber;
  }
```

```java
  //--- Input method ---
  /** read()
   *  Input:  name, age, and id number of a license
   *  Postcondition:  This License has these input values
   *                  in its instance variables.
   */
  public void read(Screen scr, Keyboard kbd) {
    scr.print("Name? "); myName = kbd.readLine();
    scr.print("Age? ");  int age = kbd.readInt();
    setAge(age);
    scr.print("Id-number? "); int id = kbd.readInt(); kbd.getChar();
    setIdNumber(id);
  }

  //--- Attribute variables ---
  private String myName;
  private int myAge;
  private int myIdNumber;
}
//----------- end of class License -----------------
```

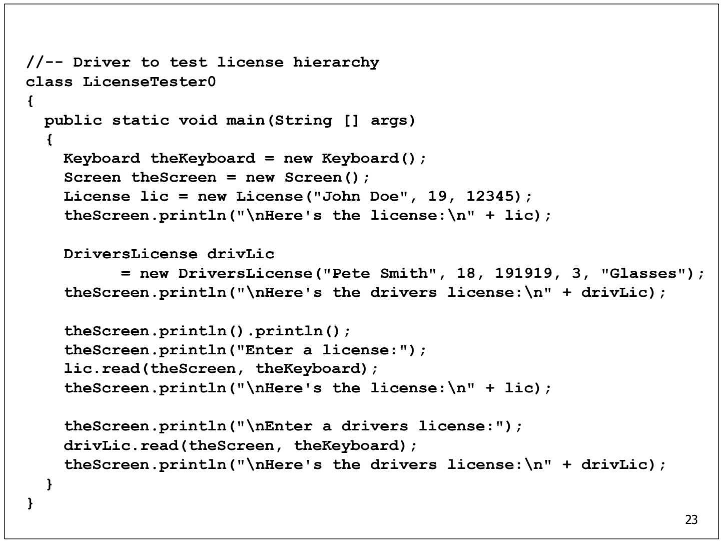
```java
/** DriversLicense.java provides a class to model drivers licenses.
 */
class DriversLicense extends License {
 //--- Constructors ---
 /** Default constructor
  *  Postcondition: A DriversLicense object is constructed with
  *                 myName == "", myAge == 0, myIdNumber == 0,
  *                 myVehicleType == 0, myRestrictionsCode == ""
  */
  public DriversLicense() {
    super();
    myVehicleType = 0;
    myRestrictionsCode = "";
  }
 /** Explicit-value constructor
  *  Receive:       int number, String name, int age, int vehicleType,
  *                 String restrictions
  *  Postcondition: A DriversLicense object is constructed with
  *                 myName == name &&  myAge && age
  *                  && myIdNumber == number
  *                  && myVehicleType == vehicleType
  *                  && myRestrictionsCode == restrictions.
  */
  public DriversLicense(String name, int age, int number,
                        int vehicleType, String restrictions) {
    super(name, age, number);
    myVehicleType = vehicleType;
    myRestrictionsCode = restrictions;
  }
```



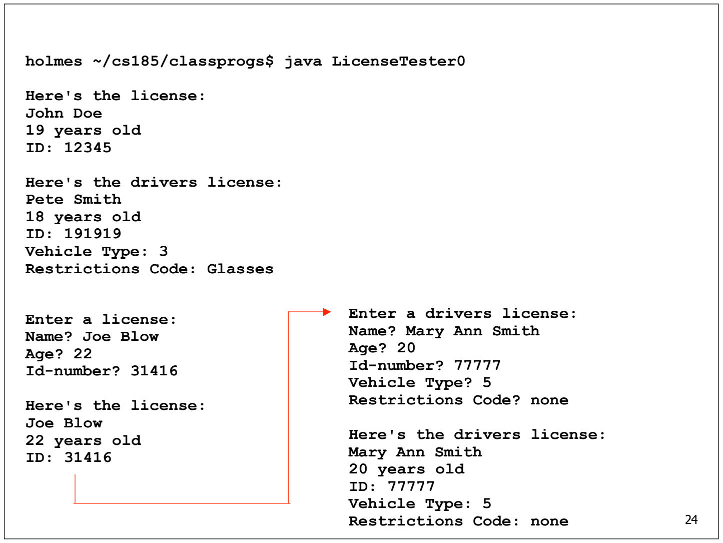
```
 //--- Accessors ---
  public int getVehicleType()  { return myVehicleType; }
  public String getRestrictionsCode()  { return myRestrictionsCode; }

 //--- Mutators --- Change public to protected if we want
 //---              these accessible only to descendants
  public void setVehicleType(int vehic)  { myVehicleType = vehic ; }
  public void setRestrictionsCode(String rc) { myRestrictionsCode = rc;

//--- Output method --- overrides toString() in License
/** toString converter
  *  Return: a String representation of a DriversLicense
  */
  public String toString() {
    return super.toString() + "\nVehicle Type: " + myVehicleType
        + "\nRestrictions Code: " + myRestrictionsCode;
  }
```



```
  //--- Input method --- overrides read() in License
  /** read()
   *  Input:  name, age, id number, vehicle type, and restrictions
   *          code of a drivers license
   *  Postcondition:  This DriversLicense has these input values
   *                  in its instance variables.
   */
  public void read(Screen scr, Keyboard kbd) {
    super.read(scr, kbd);
    scr.print("Vehicle Type? "); int vehic = kbd.readInt();
    kbd.getChar(); setVehicleType(vehic);
    scr.print("Restrictions Code? "); String rc = kbd.readLine();
    setRestrictionsCode(rc);
  }

  //--- Attribute variables ---
    private int myVehicleType;
    private String myRestrictionsCode;
}
//----------- end of class DriversLicense -----------------
```



```
//-- Driver to test license hierarchy
class LicenseTester0
{
  public static void main(String [] args)
  {
    Keyboard theKeyboard = new Keyboard();
    Screen theScreen = new Screen();
    License lic = new License("John Doe", 19, 12345);
    theScreen.println("\nHere's the license:\n" + lic);

    DriversLicense drivLic
          = new DriversLicense("Pete Smith", 18, 191919, 3, "Glasses");
    theScreen.println("\nHere's the drivers license:\n" + drivLic);

    theScreen.println().println();
    theScreen.println("Enter a license:");
    lic.read(theScreen, theKeyboard);
    theScreen.println("\nHere's the license:\n" + lic);

    theScreen.println("\nEnter a drivers license:");
    drivLic.read(theScreen, theKeyboard);
    theScreen.println("\nHere's the drivers license:\n" + drivLic);
  }
}
```



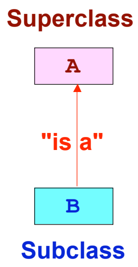
```
holmes ~/cs185/classprogs$ java LicenseTester0

Here's the license:
John Doe
19 years old
ID: 12345

Here's the drivers license:
Pete Smith
18 years old
ID: 191919
Vehicle Type: 3
Restrictions Code: Glasses


Enter a license:
Name? Joe Blow
Age? 22
Id-number? 31416

Here's the license:
Joe Blow
22 years old
ID: 31416

Enter a drivers license:
Name? Mary Ann Smith
Age? 20
Id-number? 77777
Vehicle Type? 5
Restrictions Code? none

Here's the drivers license:
Mary Ann Smith
20 years old
ID: 77777
Vehicle Type: 5
Restrictions Code: none
```

Some Properties of Inheritance

- Declaring subclasses:

```
class B extends A
{ . . . }
```

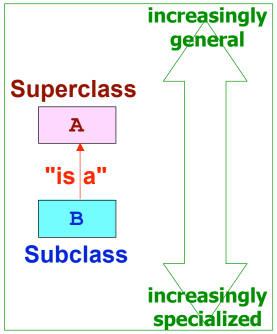
  - the "is a" relationship exists between subclass and superclass a **B** object is an **A** object
  - a unidirectional relationship: from subclass to superclass
  - Other names that help understand concept of inheritance:
    subclass: *child* class, *derived* class
    superclass: *parent* class, *base* class



- In an is-a relationship, the child must be a specialization of the parent.
- Children inherit from their parents:
  - attributes (data)
  - methods (operations)

For example, a **DriversLicense** is a **License**, so it inherits the name, age, and id attributes from **License** as well as its methods; e.g.,

```
DriversLicense d =
        new DriversLicense();
    d.setName("John Q. Doe");
```



Constructor Problem:

- A child's constructor must *initialize all attributes*, including those in the parent class.
  - Private attributes in parent classes cannot be accessed by children, even through they inherit these attributes.
  - A child may not know the details of the parent.
- Solution: The child constructor can call **super()**:

```
super(arguments_if_any);
```

  - This must be the first statement in the constructor.
  - E.g., see **DriverLicense**'s constructors.

How a Child can get at Inherited Attributes:

- Use accessor and mutator methods provided by the parent (if there are any); e.g., the accessors and mutators in **License** and **DriverLicense**
- Have the parent declare these attributes as **protected**, which allows subclasses to access them but no other classes may.

The first approach is better. A mutator can check an attempt to change an attribute and refuse it if it is invalid. If we want to restrict them to descendant classes, declare them to be protected.

The second approach would allow children and other descendants to give them invalid values.

(E.G., see constructors in **DeerHuntingLicense** class later.)

How a Child can get at Inherited Methods:

- A subclass inherits definitions of methods from its parent (and other ancestors) unless if overrides them by defining its own version.

  Example: `toString()` in `DriversLicense` overrides the version of `toString()` in `License`; similarly for `read()`.

  These inherited methods can be called by name directly within the subclass. (E.G., see constructors in `DeerHuntingLicense` class later.)
- If method `m()` in class A is overridden in subclass B, then methods in B can call the version of `m()` in A with

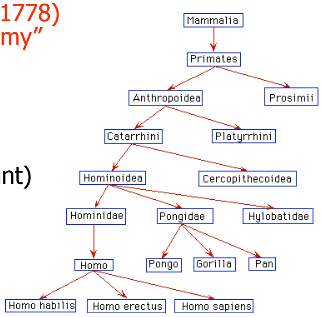
  `super.m(argument_list)`

e.g., see

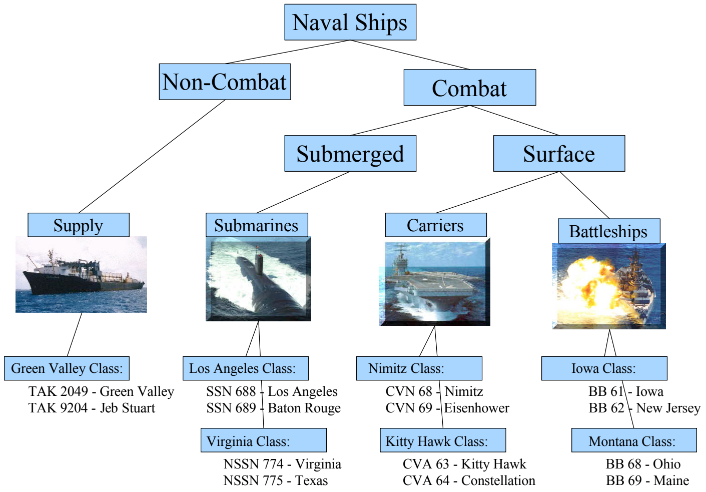
Class Hierarchies

Carl Linneaus (1707-1778)
"father of the taxonomy"

- *Systema Naturae*, 1758
- 7 levels:
  - Kingdom (e.g., animal, plant)
  - Phylum
  - Class
  - Order
  - Family
  - Genus
  - Species



Images from `www.linnean.org` & `www.kheper.auz.com`



Naval Ships

Non-Combat

Combat

Submerged

Surface

Supply

Submarines

Carriers

Battleships

Green Valley Class:
TAK 2049 - Green Valley
TAK 9204 - Jeb Stuart

Los Angeles Class:
SSN 688 - Los Angeles
SSN 689 - Baton Rouge

Nimitz Class:
CVN 68 - Nimitz
CVN 69 - Eisenhower

Iowa Class:
BB 61 - Iowa
BB 62 - New Jersey

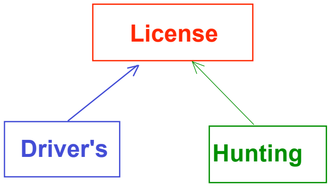
Virginia Class:
NSSN 774 - Virginia
NSSN 775 - Texas

Kitty Hawk Class:
CVA 63 - Kitty Hawk
CVA 64 - Constellation

Montana Class:
BB 68 - Ohio
BB 69 - Maine

Pictures from `http://www.warships1.com/`

Building subclasses of subclasses leads to **class hierarchies** — usually pictured as a tree but with arrows drawn from a child class to its parent class; for example, we might add another child class `HuntingLicense` to class `License`:



License

Driver's

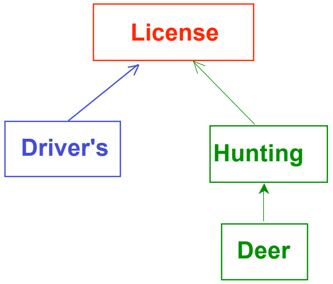
Hunting

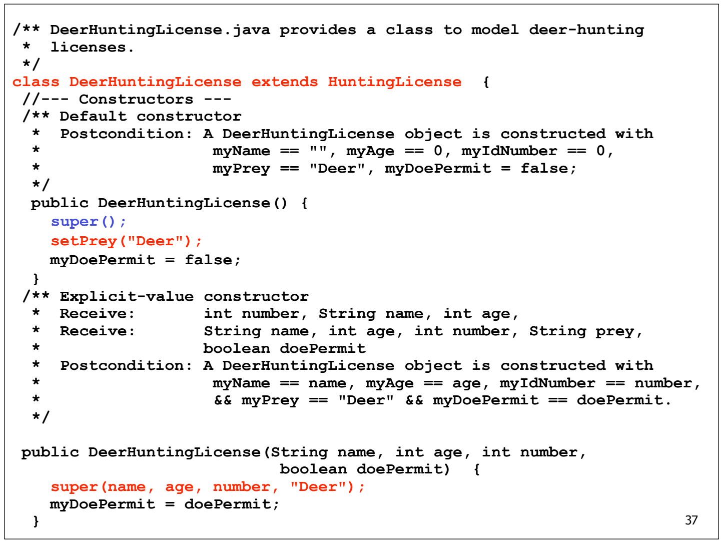
```
/** HuntingLicense.java provides a class to model hunting licenses.
 */

class HuntingLicense extends License {
 //--- Constructors ---
 /** Default constructor
  *  Postcondition: A HuntingLicense object is constructed with
  *                 myName == "", myAge == 0, myIdNumber == 0,
  *                 myPrey == "".
  */
  public HuntingLicense() {
    super();
    myPrey = "";
  }
 /** Explicit-value constructor
  *  Receive:       int number, String name, int age,
  *  Receive:       String name, int age, int number, String prey
  *  Postcondition: A HuntingLicense object is constructed with
  *                 myName == name, myAge == age, myIdNumber == number,
  *                 myPrey == prey.
  */
  public HuntingLicense(String name, int age, int number, String prey) {
    super(name, age, number);
    myPrey = prey;
  }
```

```
 //--- Accessors ---
  public String getPrey()  { return myPrey; }

 //--- Mutators --- Change public to protected if we want
 //---                 these accessible only to descendants
 public void setPrey(String prey) { myPrey = prey; }

 //--- Output method ---
 /** toString converter
  *  Return: a String representation of a HuntingLicense
  */
  public String toString() {
    return super.toString() + "\nPrey: " + myPrey;
  }
```

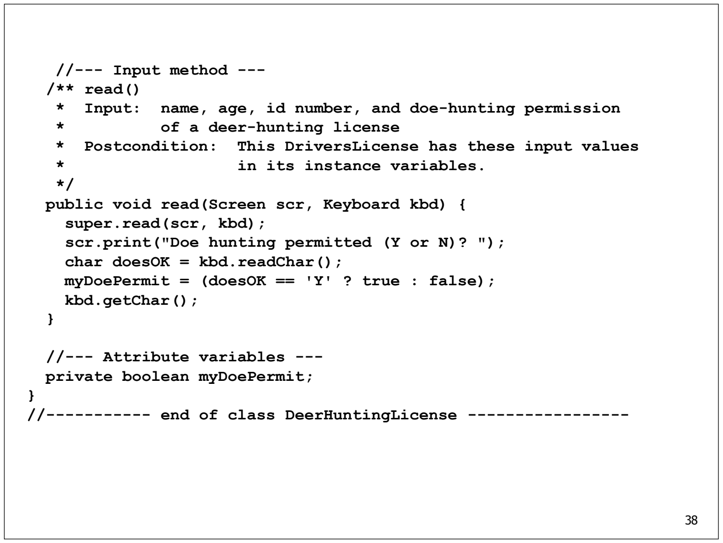
```
  //--- Input method ---
  /** read()
   *  Input:  name, age, id number, and prey of
   *          a hunting license
   *  Postcondition:  This HuntingLicense has these input values
   *                  in its instance variables.
   */
  public void read(Screen scr, Keyboard kbd) {
    super.read(scr, kbd);
    if (!myPrey.equals("Deer")) {
      scr.print("Prey? ");
      myPrey = kbd.readLine();
    }
  }

  //--- Attribute variables ---
  private String myPrey;
}
//----------- end of class HuntingLicense -----------------
```

And then we might add subclass **DeerHuntingLicense** of class **HuntingLicense**:

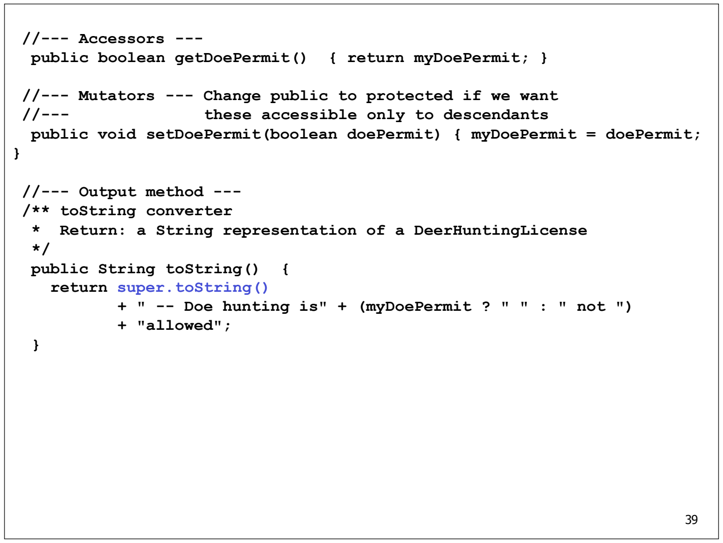




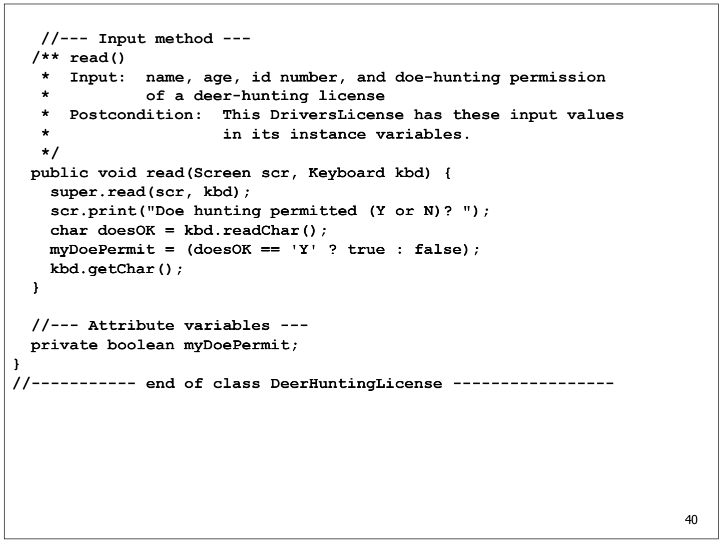
```
/** DeerHuntingLicense.java provides a class to model deer-hunting
 *  licenses.
 */
class DeerHuntingLicense extends HuntingLicense  {
 //--- Constructors ---
 /** Default constructor
  *  Postcondition: A DeerHuntingLicense object is constructed with
  *                  myName == "", myAge == 0, myIdNumber == 0,
  *                  myPrey == "Deer", myDoePermit = false;
  */
  public DeerHuntingLicense() {
    super();
    setPrey("Deer");
    myDoePermit = false;
  }
 /** Explicit-value constructor
  *  Receive:       int number, String name, int age,
  *  Receive:       String name, int age, int number, String prey,
  *                 boolean doePermit
  *  Postcondition: A DeerHuntingLicense object is constructed with
  *                  myName == name, myAge == age, myIdNumber == number,
  *                  && myPrey == "Deer" && myDoePermit == doePermit.
  */

 public DeerHuntingLicense(String name, int age, int number,
                           boolean doePermit)  {
    super(name, age, number, "Deer");
    myDoePermit = doePermit;
  }
```



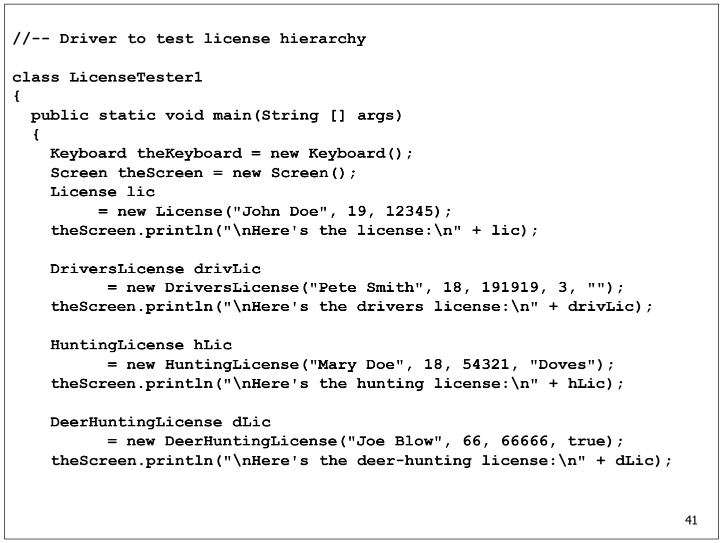
```
   //--- Input method ---
  /** read()
   *  Input:  name, age, id number, and doe-hunting permission
   *          of a deer-hunting license
   *  Postcondition:  This DriversLicense has these input values
   *                  in its instance variables.
   */
  public void read(Screen scr, Keyboard kbd) {
    super.read(scr, kbd);
    scr.print("Doe hunting permitted (Y or N)? ");
    char doesOK = kbd.readChar();
    myDoePermit = (doesOK == 'Y' ? true : false);
    kbd.getChar();
  }

  //--- Attribute variables ---
  private boolean myDoePermit;
}
//----------- end of class DeerHuntingLicense -----------------
```



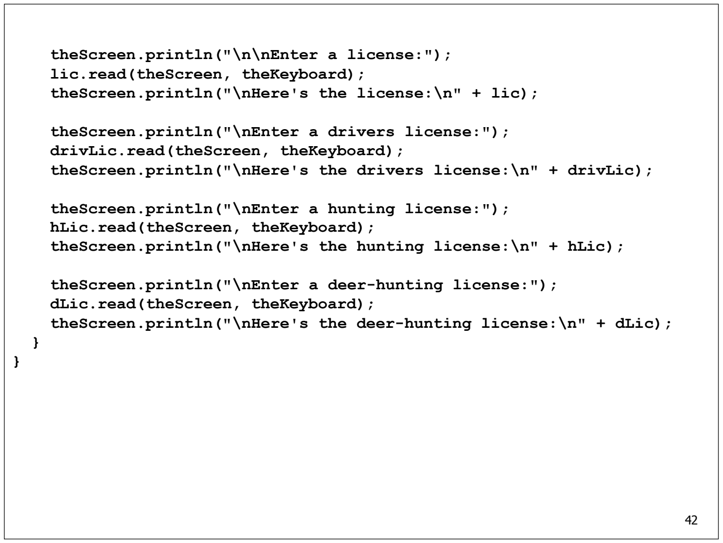
```
 //--- Accessors ---
  public boolean getDoePermit()  { return myDoePermit; }

 //--- Mutators --- Change public to protected if we want
 //---                 these accessible only to descendants
  public void setDoePermit(boolean doePermit) { myDoePermit = doePermit;
}

 //--- Output method ---
 /** toString converter
  *  Return: a String representation of a DeerHuntingLicense
  */
  public String toString()  {
    return super.toString()
           + " -- Doe hunting is" + (myDoePermit ? " " : " not ")
           + "allowed";
  }
```



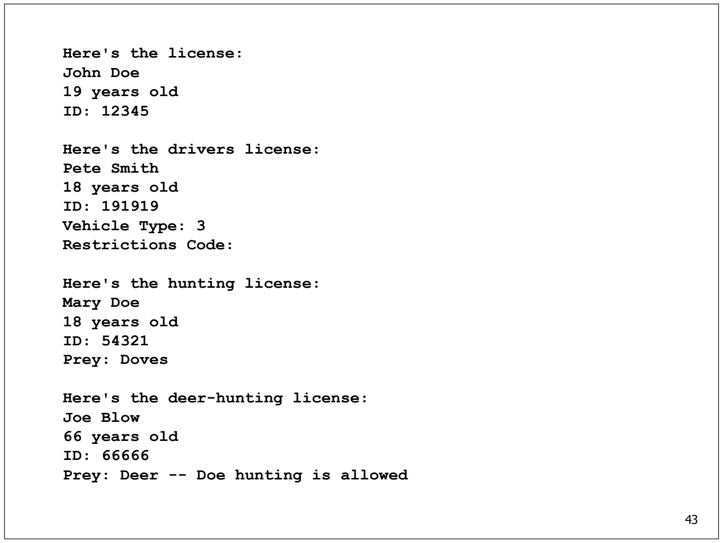
```
   //--- Input method ---
  /** read()
   *  Input:  name, age, id number, and doe-hunting permission
   *          of a deer-hunting license
   *  Postcondition:  This DriversLicense has these input values
   *                  in its instance variables.
   */
  public void read(Screen scr, Keyboard kbd) {
    super.read(scr, kbd);
    scr.print("Doe hunting permitted (Y or N)? ");
    char doesOK = kbd.readChar();
    myDoePermit = (doesOK == 'Y' ? true : false);
    kbd.getChar();
  }

  //--- Attribute variables ---
  private boolean myDoePermit;
}
//----------- end of class DeerHuntingLicense -----------------
```



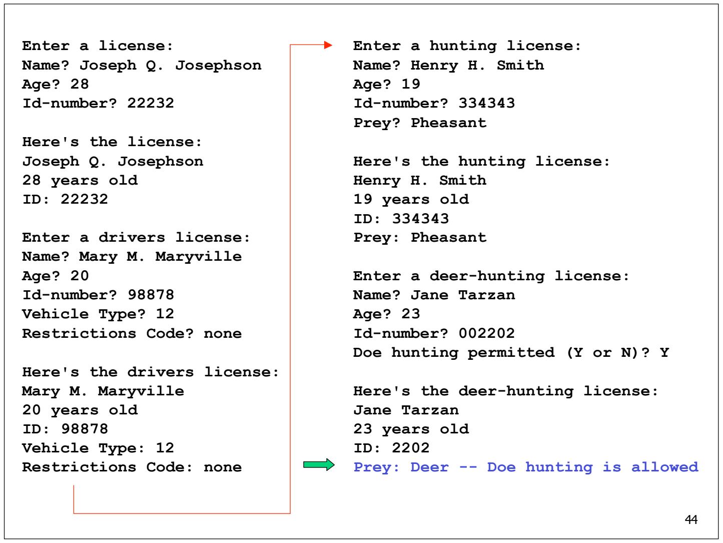
```
//-- Driver to test license hierarchy

class LicenseTester1
{
  public static void main(String [] args)
  {
    Keyboard theKeyboard = new Keyboard();
    Screen theScreen = new Screen();
    License lic
         = new License("John Doe", 19, 12345);
    theScreen.println("\nHere's the license:\n" + lic);

    DriversLicense drivLic
          = new DriversLicense("Pete Smith", 18, 191919, 3, "");
    theScreen.println("\nHere's the drivers license:\n" + drivLic);

    HuntingLicense hLic
          = new HuntingLicense("Mary Doe", 18, 54321, "Doves");
    theScreen.println("\nHere's the hunting license:\n" + hLic);

    DeerHuntingLicense dLic
          = new DeerHuntingLicense("Joe Blow", 66, 66666, true);
    theScreen.println("\nHere's the deer-hunting license:\n" + dLic);
```



```
    theScreen.println("\n\nEnter a license:");
    lic.read(theScreen, theKeyboard);
    theScreen.println("\nHere's the license:\n" + lic);

    theScreen.println("\nEnter a drivers license:");
    drivLic.read(theScreen, theKeyboard);
    theScreen.println("\nHere's the drivers license:\n" + drivLic);

    theScreen.println("\nEnter a hunting license:");
    hLic.read(theScreen, theKeyboard);
    theScreen.println("\nHere's the hunting license:\n" + hLic);

    theScreen.println("\nEnter a deer-hunting license:");
    dLic.read(theScreen, theKeyboard);
    theScreen.println("\nHere's the deer-hunting license:\n" + dLic);
  }
}
```



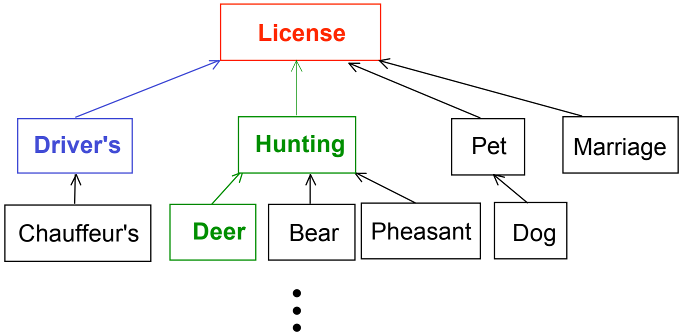
```
Here's the license:
John Doe
19 years old
ID: 12345

Here's the drivers license:
Pete Smith
18 years old
ID: 191919
Vehicle Type: 3
Restrictions Code:

Here's the hunting license:
Mary Doe
18 years old
ID: 54321
Prey: Doves

Here's the deer-hunting license:
Joe Blow
66 years old
ID: 66666
Prey: Deer -- Doe hunting is allowed
```



```
Enter a license:
Name? Joseph Q. Josephson
Age? 28
Id-number? 22232

Here's the license:
Joseph Q. Josephson
28 years old
ID: 22232

Enter a drivers license:
Name? Mary M. Maryville
Age? 20
Id-number? 98878
Vehicle Type? 12
Restrictions Code? none

Here's the drivers license:
Mary M. Maryville
20 years old
ID: 98878
Vehicle Type: 12
Restrictions Code: none
```

```
Enter a hunting license:
Name? Henry H. Smith
Age? 19
Id-number? 334343
Prey? Pheasant

Here's the hunting license:
Henry H. Smith
19 years old
ID: 334343
Prey: Pheasant

Enter a deer-hunting license:
Name? Jane Tarzan
Age? 23
Id-number? 002202
Doe hunting permitted (Y or N)? Y

Here's the deer-hunting license:
Jane Tarzan
23 years old
ID: 2202
Prey: Deer -- Doe hunting is allowed
```

and so on . . .



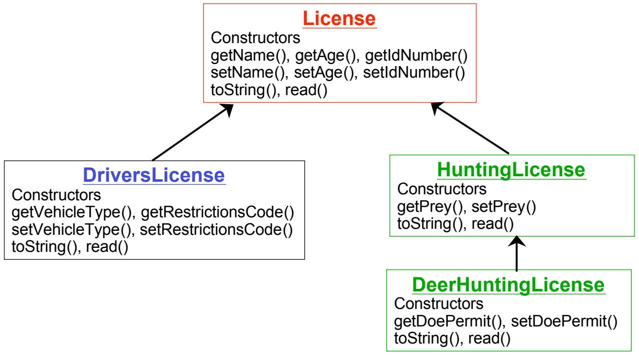
- The class at the top of the hierarchy is called the root class.
- Fundamental Property of Inheritance: *All non-root classes inherit all the attributes and operations of ever ancestor class.*
- This means that all of the attributes and operations of every class can be *reused in every descendant class.*

Java's Class Hierarchy

- All classes in Java must fit into the Java *class hierarchy*.
  - http://java.sun.com/j2se/1.4.1/docs/api/
  - By default, a class inherits from `Object`, the root of the class hierarchy. For example, it inherits definitions of `toString()`, `equals()`, and `getClass()`.
  - There are currently over 1600 classes.
- What's the point?
  - To save time coding features that are common to many applications/classes.

More Properties of Inheritance

To illustrate the properties, we'll focus on just the following part of our License hierarchy:

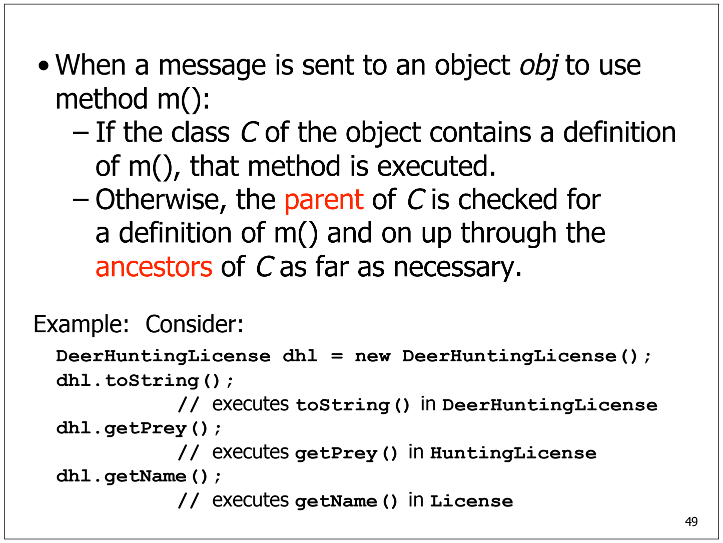


License
Constructors
getName(), getAge(), getIdNumber()
setName(), setAge(), setIdNumber()
toString(), read()

DriversLicense
Constructors
getVehicleType(), getRestrictionsCode()
setVehicleType(), setRestrictionsCode()
toString(), read()

HuntingLicense
Constructors
getPrey(), setPrey()
toString(), read()

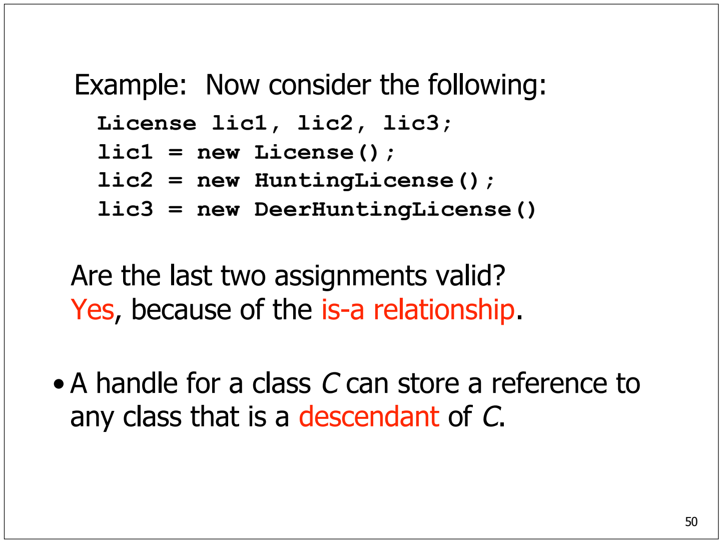
DeerHuntingLicense
Constructors
getDoePermit(), setDoePermit()
toString(), read()



- When a message is sent to an object *obj* to use method m():
  - If the class *C* of the object contains a definition of m(), that method is executed.
  - Otherwise, the parent of *C* is checked for a definition of m() and on up through the ancestors of *C* as far as necessary.

Example: Consider:

```
DeerHuntingLicense dhl = new DeerHuntingLicense();
dhl.toString();
            // executes toString() in DeerHuntingLicense
dhl.getPrey();
            // executes getPrey() in HuntingLicense
dhl.getName();
            // executes getName() in License
```

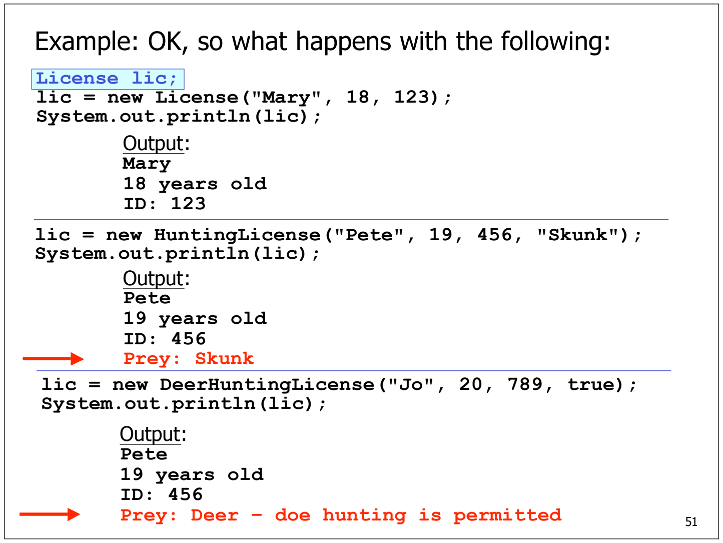


Example: Now consider the following:

```
License lic1, lic2, lic3;
lic1 = new License();
lic2 = new HuntingLicense();
lic3 = new DeerHuntingLicense()
```

Are the last two assignments valid?
Yes, because of the is-a relationship.

- A handle for a class *C* can store a reference to any class that is a descendant of *C*.

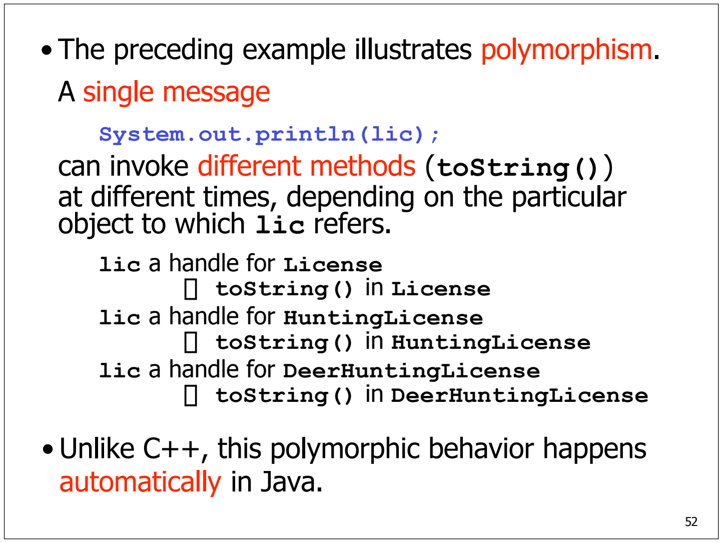


Example: OK, so what happens with the following:

```
License lic;
lic = new License("Mary", 18, 123);
System.out.println(lic);
         Output:
         Mary
         18 years old
         ID: 123
```

```
lic = new HuntingLicense("Pete", 19, 456, "Skunk");
System.out.println(lic);
         Output:
         Pete
         19 years old
         ID: 456
  →      Prey: Skunk
```

```
lic = new DeerHuntingLicense("Jo", 20, 789, true);
System.out.println(lic);
         Output:
         Pete
         19 years old
         ID: 456
  →      Prey: Deer - doe hunting is permitted
```



- The preceding example illustrates polymorphism.
  A single message

  `System.out.println(lic);`

  can invoke different methods (`toString()`) at different times, depending on the particular object to which `lic` refers.

```
lic a handle for License
        □ toString() in License
lic a handle for HuntingLicense
        □ toString() in HuntingLicense
lic a handle for DeerHuntingLicense
        □ toString() in DeerHuntingLicense
```

- Unlike C++, this polymorphic behavior happens automatically in Java.

Abstract Methods and Classes

Suppose we had decided to add a fee attribute to licenses. The problem is that the way fees are computed varies with the kind of license; for example, a driver's license may have a flat fee of $25.55 but a deer-hunting license might be $20.00 without a doe permit but $30.00 with a doe permit.

One approach would be to define some generic default `getFee()` method for the `License` class and let the descendant classes override it as necessary.

Another approach: Make `getFee()` an abstract method in the `License` class, which makes License an abstract class:

```
abstract class License extends Object
{
  . . .
  abstract double getFee();
  . . .
}
```

- There can be no instances of an abstract class (because there is no definition given for the abstract method).

  Example: `License lic = new License();` causes a compiler error.

- Every subclass must either provide a definition of this abstract method; or it must inherit it and must then be declared to be an abstract subclass.

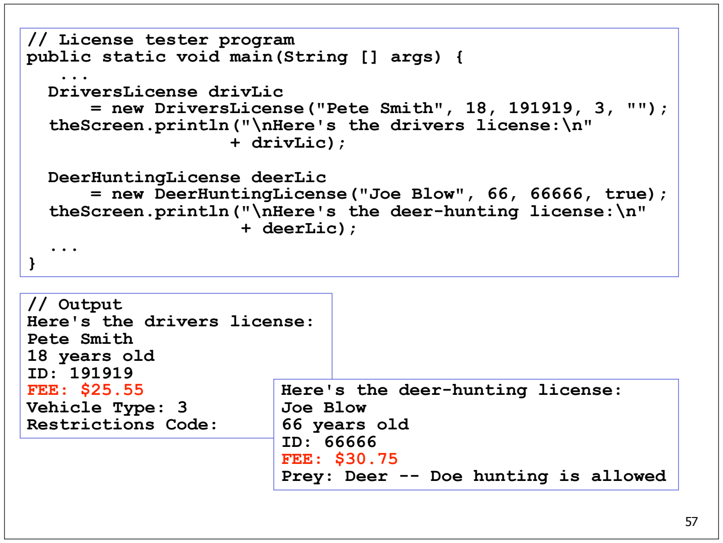
Example:

```
abstract class License extends Object {
  . . .
  public String toString() {
    return myName   + "\n" + myAge + " years old"
                    + "\nID:  " + myIdNumber
                    + "\nFEE: $" + getFee();
  }
  . . .
}
```

```
class DriversLicense extends License {
  . . .
  public double getFee() { return 25.55; }
  . . .
}
```

```
// Abstract class because different kinds of
// hunting licenses have different fee structures
abstract class HuntingLicense extends License {
  . . .
}
```

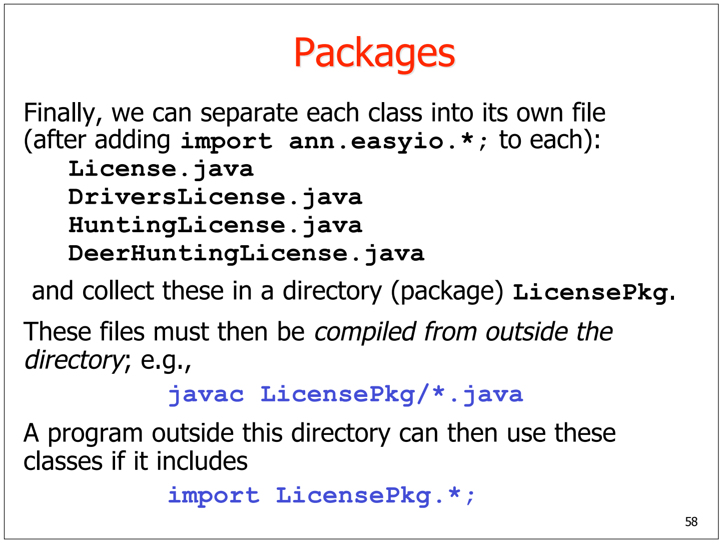
```
class DeerHuntingLicense extends HuntingLicense {
  . . .
  public double getFee() {
    final double DEER_FEE = 20.00,
                 DOE_FEE = 10.75;
    if (myDoePermit)
      return DEER_FEE + DOE_FEE;
    //else
      return DEER_FEE;
  }
  . . .
}
```



```
// License tester program
public static void main(String [] args) {
    ...
  DriversLicense drivLic
      = new DriversLicense("Pete Smith", 18, 191919, 3, "");
  theScreen.println("\nHere's the drivers license:\n"
                   + drivLic);

  DeerHuntingLicense deerLic
      = new DeerHuntingLicense("Joe Blow", 66, 66666, true);
  theScreen.println("\nHere's the deer-hunting license:\n"
                    + deerLic);
   ...
}
```

```
// Output
Here's the drivers license:
Pete Smith
18 years old
ID: 191919
FEE: $25.55
Vehicle Type: 3
Restrictions Code:
```

```
Here's the deer-hunting license:
Joe Blow
66 years old
ID: 66666
FEE: $30.75
Prey: Deer -- Doe hunting is allowed
```



Packages

Finally, we can separate each class into its own file (after adding **`import ann.easyio.*;`** to each):

```
    License.java
    DriversLicense.java
    HuntingLicense.java
    DeerHuntingLicense.java
```

and collect these in a directory (package) **`LicensePkg`**.

These files must then be *compiled from outside the directory*; e.g.,

```
javac LicensePkg/*.java
```

A program outside this directory can then use these classes if it includes

```
import LicensePkg.*;
```