Java Advanced

Rahul Deodhar
rahuldeodhar@gmail.com
www.rahuldeodhar.com
+91 9820213813

Chapter 6 - Object-Oriented Programming

- Object-oriented programming overview
  - objects
  - classes
  - encapsulation
- UML Class Diagram
- First OOP Class
- `private` and `public` Access
- Driver Class
- Reference Variables and Instantiation
- Calling a Method
- Calling Object

Chapter 6 - Object-Oriented Programming

- The `this` Reference
- Default Values
- Variable Persistence
- OOP Tracing Procedure (hidden)
- UML Class Diagram for Next Version of the Mouse Program
- Local Variables
- `return` statement
- `void` Return Type
- Empty `return` Statement
- Argument Passing
- Specialized methods:
  - accessor methods
  - mutator methods
  - `boolean` methods

Object-Oriented Programming Overview

- In the old days, the standard programming technique was called "procedural programming."
- That's because the emphasis was on the procedures or tasks that made up a program.
- You'd design your program around what you thought were the key procedures.
- Today, the most popular programming technique is *object-oriented programming* (OOP).
- With OOP, instead of thinking first about procedures, you think first about the things in your problem. The things are called objects.

Object-Oriented Programming Overview

- An object is:

  A set of related data which identifies the current *state* of the object.

  + a set of behaviors

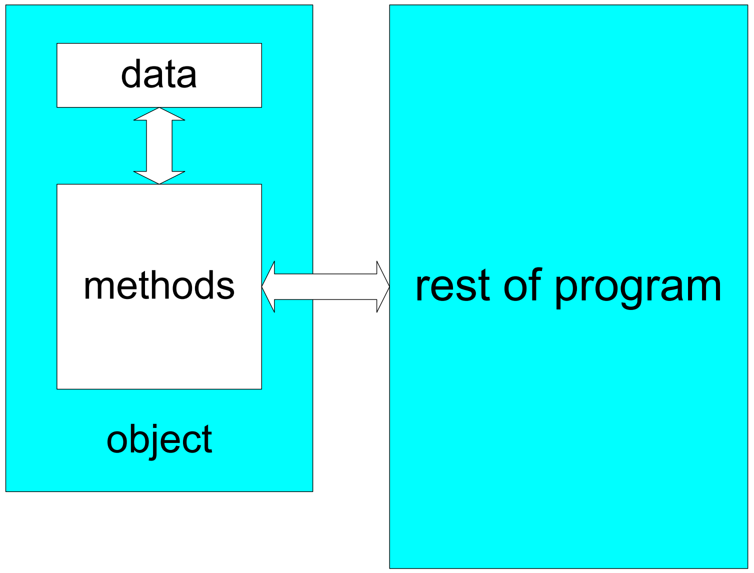
- Example objects:

human entities	physical objects	mathematical entities
employees	cars in a traffic-flow simulation	points in a coordinate system
customers	aircraft in an air-traffic control system	complex numbers
students	electrical components in a circuit-design program	time

- Car object in a traffic-flow simulation:
  - data = ?
  - methods = ?

Object-Oriented Programming Overview

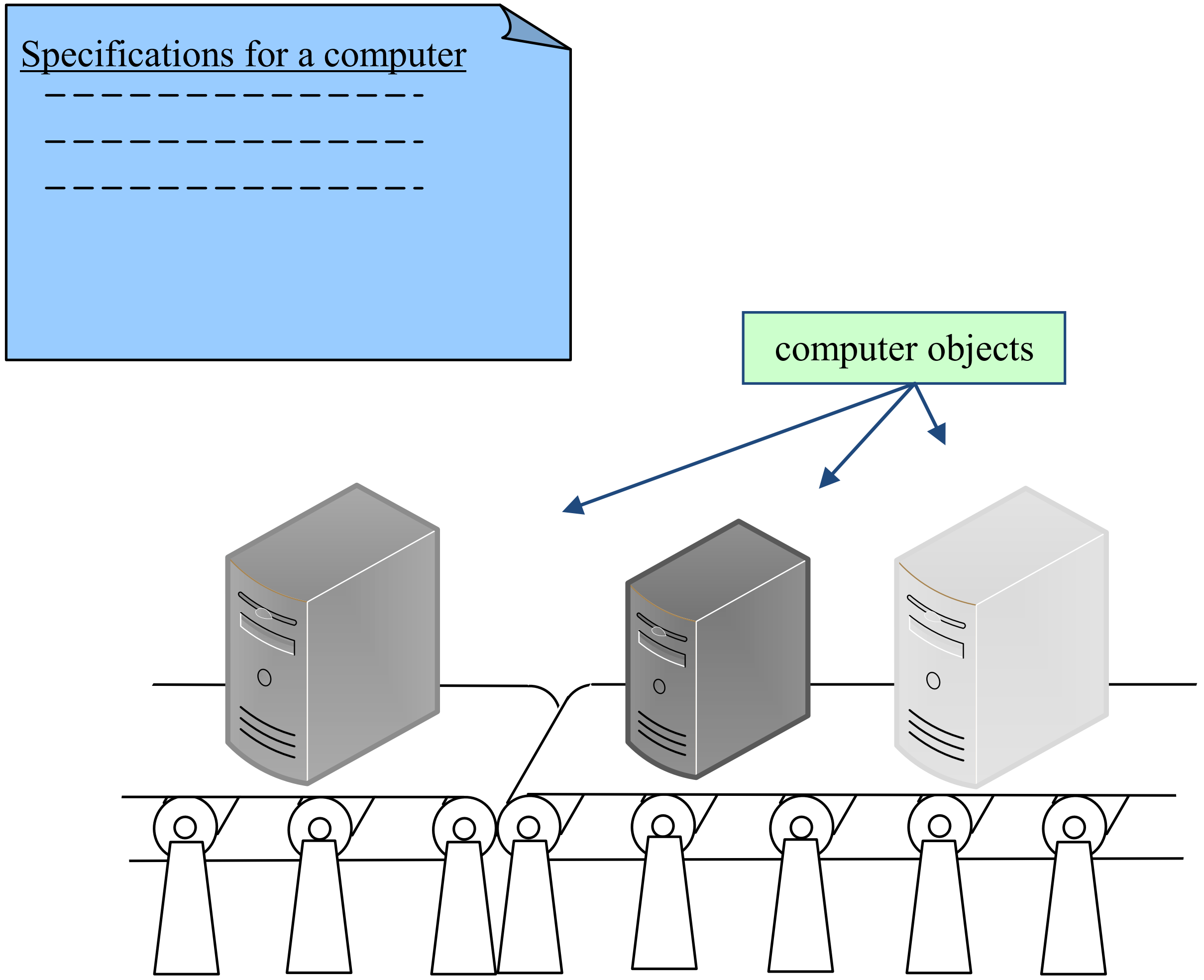
- Benefits of OOP:
  - Programs are more understandable -
    - Since people tend to think about problems in terms of objects, it's easier for people to understand a program that's split into objects.
  - Fewer errors -
    - Since objects provide *encapsulation* (isolation) for the data, it's harder for the data to get messed up.



Object-Oriented Programming Overview

- A *class* is a description for a set of objects.
- On the next slide, note the three computers on a conveyer belt in a manufacturing plant:
  - The three computers represent objects, and the specifications document represents a class. The specifications document is a blueprint that describes the computers: it lists the computers' components and describes the computers' features.
- Think of an object as a physical example for a class's description. More formally, we say that an object is an *instance* of a class.

Object-Oriented Programming Overview



Object-Oriented Programming Overview

- A *class* is a description for a set of objects. The description consists of:

    a list of variables
  + a list of methods

- Classes can define two types of variables – instance variables and class variables. And classes can define two types of methods – instance methods and class methods. Instance variables and instance methods are more common than class variables and class methods, and we'll focus on instance variables and instance methods in this chapter and the next several chapters.

Object-Oriented Programming Overview

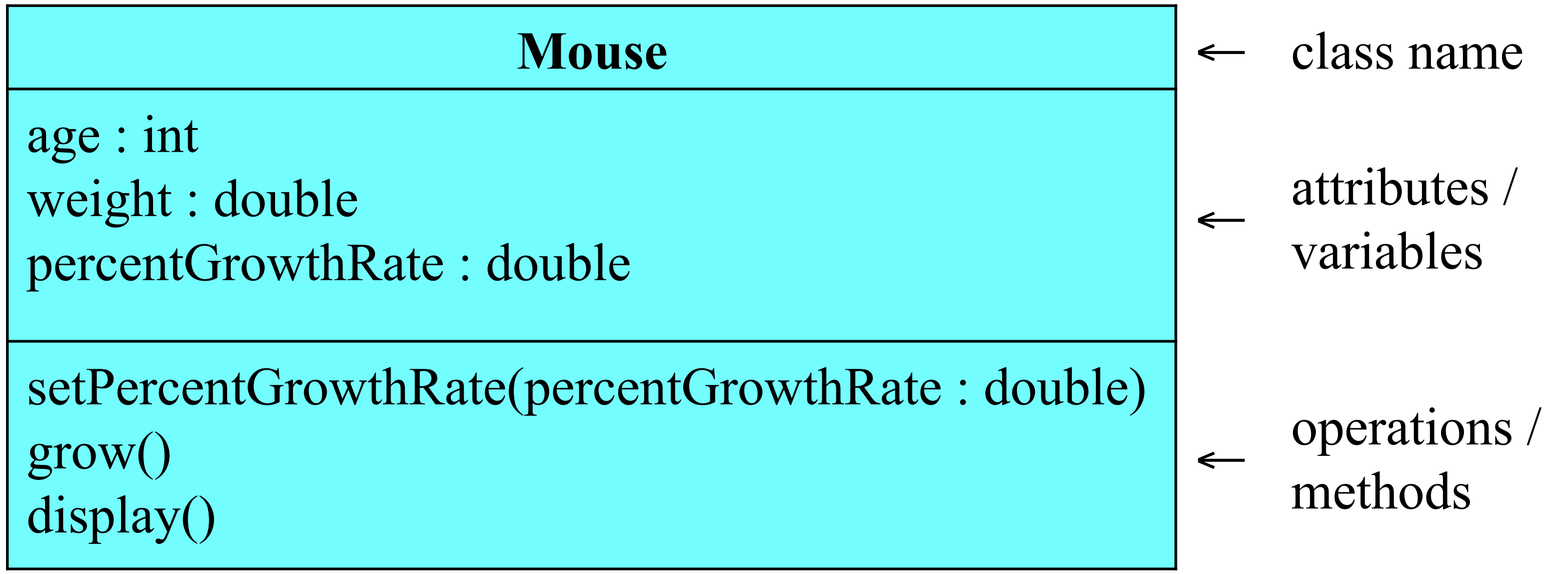
- A class's *instance variables* specify the type of data that an object can store.
  - For example, if you have a class for computer objects, and the `Computer` class contains a `hardDiskSize` instance variable, then each computer object stores a value for the size of the computer's hard disk.
- A class's *instance methods* specify the behavior that an object can exhibit.
  - For example, if you have a class for computer objects, and the `Computer` class contains a `printSpecifications` instance method, then each computer object can print a specifications report (the specifications report shows the computer's hard disk size, CPU speed, cost, etc.).

Object-Oriented Programming Overview

- Note the use of the term “instance” in “instance variable” and “instance method.” That reinforces the fact that instance variables and instance methods are associated with a particular object instance. For example, each computer object has its own value for the `hardDiskSize` instance variable.
- That contrasts with class variables and class methods, which you saw in Chapter 5. Class variables and class methods are associated with an entire class. For example, the `Math` class contains the `PI` class variable and the `round` class method. `PI` and `round` are associated with the entire `Math` class, not with a particular instance of the `Math` class. We'll cover class variables and class methods in more detail in Chapter 9.

UML Class Diagram

- UML:
  - Stands for Unified Modeling Language.
  - It's a diagrammatic methodology for describing classes, objects, and the relationships between them.
  - It is widely accepted in the software industry as a standard for modeling OOP designs.
- Example:
  - UML class diagram for a `Mouse` class:



First OOP Class

```
/*************************************************************
* Mouse.java
* Dean & Dean
*
* This class models a mouse for a growth simulation program.
*************************************************************/

public class Mouse
{
  private int age = 0;                  // age of mouse in days
  private double weight = 1.0;          // weight of mouse in grams
  private double percentGrowthRate;     // % weight increase per day

  //**********************************************************

  // This method assigns the mouse's percent growth rate.

  public void setPercentGrowthRate(double percentGrowthRate)
  {
    this.percentGrowthRate = percentGrowthRate;
  } // end setPercentGrowthRate
```

instance variable declarations

parameter

method body

To access instance variables, use `this` dot.

First OOP Class

```
  //***********************************************************

  // This method simulates one day of growth for the mouse.

  public void grow()
  {
    this.weight += (.01 * this.percentGrowthRate * this.weight);
    this.age++;
  } // end grow

  //***********************************************************

  // This method prints the mouses's age and weight.

  public void display()
  {
    System.out.printf(
      "Age = %d, weight = %.3f\n", this.age, this.weight);
  } // end display
} // end class Mouse
```

`private` and `public` Access

- `private` and `public` are *access modifiers*. When you apply an access modifier to a member of a class, you determine how easy it is for the member to be accessed. *Accessing* a member refers to either reading the member's value or modifying it.
- If you declare a member to be `private`, then the member can be accessed only from within the member's class. Instance variables are almost always declared with the `private` modifier because you almost always want an object's data to be hidden. Making the data hard to access is what encapsulation is all about and it's one of the cornerstones of OOP.
- If you declare a member to be `public`, then the member can be accessed from anywhere (from within the member's class, and also from outside the member's class). Methods are usually declared with the `public` modifier because you normally want to be able to call them from anywhere.

Driver Class

```
/****************************************
* MouseDriver.java
* Dean & Dean
*
* This is a driver for the Mouse class.
****************************************/

import java.util.Scanner;

public class MouseDriver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    double growthRate;
    Mouse gus = new Mouse();
    Mouse jaq = new Mouse();
```

Driver Class

```
    System.out.print("Enter growth rate as a percentage: ");
    growthRate = stdIn.nextDouble();
    gus.setPercentGrowthRate(growthRate);
    jaq.setPercentGrowthRate(growthRate);

    gus.grow();
    jaq.grow();
    gus.grow();
    gus.display();
    jaq.display();
  } // end main
} // end class MouseDriver
```

Reference Variables and Instantiation

- To declare a *reference variable* (which holds the address in memory where an object is stored):

  <class-name> *<reference-variable>*`;`

- To instantiate/create an object and assign its address into a reference variable:

  <reference-variable> `= new` *<class-name>*`()`

- Example code:

```
Mouse gus;
gus = new Mouse();
```

declaration

instantiation

- This single line is equivalent to the above two lines:

```
Mouse gus = new Mouse();
```

initialization

Calling a Method

- After instantiating an object and assigning its address into a reference variable, call/invoke an instance method using this syntax:

  <reference-variable>`.`*<method-name>*`(`*<comma-separated-arguments>*`);`

- Here are three example instance method calls from the `MouseDriver` class:

```
gus.setPercentGrowthRate(growthRate);
gus.grow();
gus.display();
```

Calling Object

- A *calling object* is the object that appears at the left of the dot in a call to an instance method.
- Can you find the calling objects below?

```
public static void main(String[] args)
{
  Scanner stdIn = new Scanner(System.in);
  double growthRate;
  Mouse gus = new Mouse();

  System.out.print("Enter growth rate as a percentage: ");
  growthRate = stdIn.nextDouble();
  gus.setPercentGrowthRate(growthRate);
  gus.grow();
  gus.display();
} // end main
```

The `this` Reference

- The `this` *reference*:
  - When used in conjunction with a dot and an instance variable, "this" is referred to as the *this reference*. Note this example from the `Mouse` class's `grow` method:

    ```
    this.weight += (.01 * this.percentGrowthRate * this.weight);
    ```

  - The `this` reference is used inside of a method to refer to the object that called the method; in other words, the `this` reference refers to the calling object.

- So what's so great about having a special name for the calling object inside of a method? Why not just use the original name, `gus` or `jaq`, inside the method?
  - Because if the original name were to be used, then the method would only work for the one specified calling object. By using a generic name (`this`) for the calling object, then the method is more general purpose. For example, by using `this`, the `grow` method is able to specify weight gain for any `Mouse` object that calls it. If `gus` calls `grow`, then `gus`'s weight is updated, whereas if `jaq` calls `grow`, then `jaq`'s weight is updated.

Default Values

- A variable's *default value* is the value that the variable gets if there's no explicit initialization.
- `Mouse` class's instance variable declarations:

```
private int age = 0;
private double weight = 1.0;
private double percentGrowthRate;
```

explicit initializations

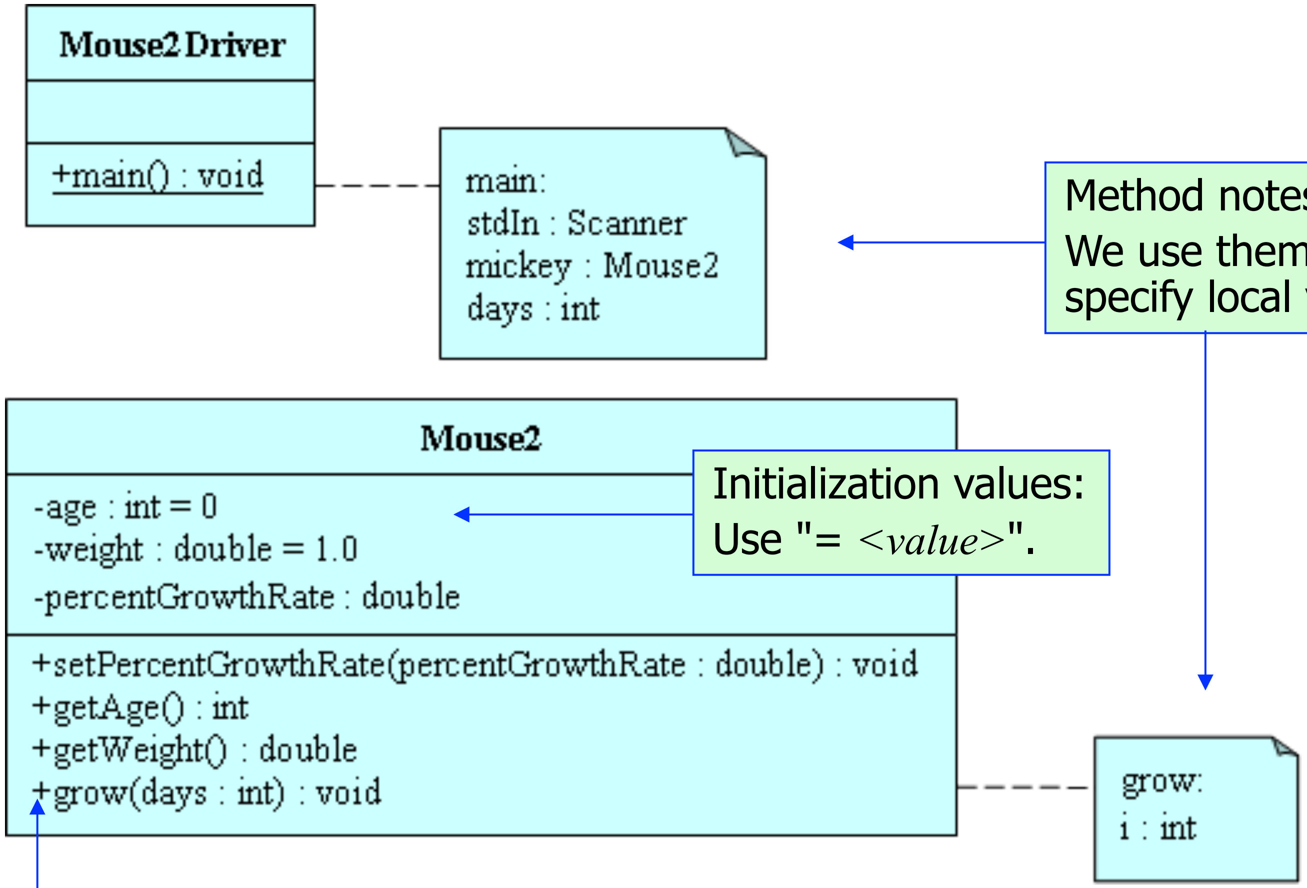
`percentGrowthRate` gets default value of 0.0

- Here are the default values for instance variables:
  - integer types get 0
  - floating point types get 0.0
  - `boolean` types get `false`
  - reference types get `null`
    - Note that a `String` is a reference type so it gets `null` by default.

Variable Persistence

- A variable's *persistence* is how long a variable's value survives before it's wiped out.
- Instance variables persist for the duration of a particular object. Thus, if an object makes two method calls, the second called method does not reset the calling object's instance variables to their initialized values. Instead, the object's instance variables retain their values from one method call to the next.

UML Class Diagram for Next Version of the Mouse Program



Method notes:
We use them here to specify local variables.

Initialization values:
Use "= <*value*>".

Member accessibility:
Use "-" for `private` access and "+" for `public` access.

Local Variables

- A *local variable* is a variable that's declared inside a method. That's different from an instance variable, which is declared at the top of a class, outside all the methods.
- A local variable is called "local" because it can be used only inside of the method in which it is declared – it is completely local to the method.
- In the `Mouse2Driver` class on the next slide, note how the `main` method has three local variables - `stdIn`, `mickey`, and `days`. And in the `Mouse2` class, note how the `grow` method has one local variable - `i`.

Mouse2Driver Class

```
import java.util.Scanner;

public class Mouse2Driver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Mouse2 mickey = new Mouse2();
    int days;

    mickey.setPercentGrowthRate(10);
    System.out.print("Enter number of days to grow: ");
    days = stdIn.nextInt();
    mickey.grow(days);
    System.out.printf("Age = %d, weight = %.3f\n",
      mickey.getAge(), mickey.getWeight());
  } // end main
} // end class Mouse2Driver
```

local variables

`Mouse2` Class

```
import java.util.Scanner;

public class Mouse2
{
  private int age = 0;                  // age in days
  private double weight = 1.0;          // weight in grams
  private double percentGrowthRate;   // % daily weight gain

  //**********************************************************

  public void setPercentGrowthRate(double percentGrowthRate)
  {
    this.percentGrowthRate = percentGrowthRate;
  } // end setPercentGrowthRate

  //**********************************************************

  public int getAge()
  {
    return this.age;
  } // end getAge
```

Mouse2 Class

```
  //*********************************************************

  public double getWeight()
  {
    return this.weight;
  } // end getWeight

  //*********************************************************

  public void grow(int days)
  {
    for (int i=0; i<days; i++)
    {
      this.weight +=
        (.01 * this.percentGrowthRate * this.weight);
    }
    this.age += days;
  } // end grow
} // end class Mouse2
```

local variable

`return` Statement

- The `return` statement allows you to pass a value from a method back to the place where the method was called. Note the following example.
- From the `Mouse2` class:

```
public int getAge()
{
  return this.age;
} // end getAge
```

return type

return statement

- From the `Mouse2Driver` class:

```
System.out.printf("Age = %d, weight = %.3f\n",
  mickey.getAge(), mickey.getWeight());
```

method call

- Note the *return type* in the above example. It has to match the type of the value that's being returned in the `return` statement.

`void` Return Type

- As shown in the below `grow` method from the `Mouse2` class, if a method does not return a value, then the method must specify `void` for its return type.

```
public void grow(int days)
{
  for (int i=0; i<days; i++)
  {
    this.weight +=
      (0.01 * this.percentGrowthRate * this.weight);
  }
  this.age += days;
} // end grow
```

Empty `return` Statement

- For methods with a `void` return type, it's legal to have an empty `return` statement. The empty `return` statement looks like this:

```
return;
```

- The empty `return` statement does what you'd expect:
  - It terminates the current method and causes control to be passed to the calling module at the point that immediately follows the method call that called the current method.

Empty `return` Statement

- Suppose you'd like to model mouse growth only up through mouse adolescence. This `grow` method does that by stopping a mouse's growth after 100 days:

```
public void grow(int days)
{
  int endAge;
  endAge = this.age + days;
  while (this.age < endAge)
  {
    if (this.age >= 100)
    {
      return;
    }
    this.weight +=
      .01 * this.percentGrowthRate * this.weight;
    this.age++;
  }  // end while
} // end grow
```

empty `return` statement

Empty `return` Statement

- Code that uses an empty `return` statement(s) can always be replaced by code that does not use the empty `return` statement(s). For example, here's a return-less version of the previous `grow` method:

```
public void grow(int days)
{
  int endAge;
  endAge = this.age + days;
  if (endAge > 100)
  {
    endAge = 100;
  }
  while (this.age < endAge)
  {
    this.weight +=
      (.01 * this.percentGrowthRate * this.weight);
    this.age++;
  }  // end while
} // end grow
```

`return` Statement Within a Loop

- Software engineering observation:
  - Real-world programmers are often asked to maintain (fix and improve) other people's code. In doing that, they oftentimes find themselves having to examine the loops and, even more specifically, the loop termination conditions in the program they're working on. Therefore, it's important that the loop termination conditions are clear. Normally, loop termination conditions appear in standard places: `while` loop heading, `do` loop closing, `for` loop heading's condition part. However, in using a `return` statement inside a loop, the `return` statement introduces a loop termination that's not in one of the standard places. For example, in the `grow` method two slides ago, the `return` statement is "hidden" inside an `if` statement that's embedded in a `while` loop.
  - In the interest of maintainability, you should use restraint when considering the use of a `return` statement inside of a loop. Based on the context, if inserting a `return` statement(s) inside a loop improves clarity, then feel free to insert. However, if it simply makes the coding chores easier and it does not add clarity, then don't insert.

Local Variable Default Values

- The default value for a local variable is *garbage*.
- Garbage means that the variable's value is unknown - it's whatever just happens to be in memory at the time that the variable is created.
- When doing a trace, use a "?" to indicate garbage.
- If a program attempts to access a variable that contains garbage, the compiler generates an error. For example, what happens when the following method is compiled?

```
public void grow(int days)
{
  for (int i; i<days; i++)
  {
    this.weight +=
      (.01 * this.percentGrowthRate * this.weight);
  }
  this.age += days;
} // end grow
```

- Since `i` is no longer assigned zero, `i` contains garbage when the `i<days` condition is tested. That causes the compiler to generate this error message:

```
variable i might not have been initialized
```

Local Variable Persistence

- Local variables persist only for the duration of the method (or `for` loop) in which the local variable is defined. The next time the method (or `for` loop) is executed, the local variable's value resets to its initial value.

Argument Passing

- What is the output for the following `Mouse3Driver` and `Mouse3` classes?

```
public class Mouse3Driver
{
  public static void main(String[] args)
  {
    Mouse3 minnie = new Mouse3();
    int days = 365;
    minnie.grow(days);
    System.out.println("# of days aged = " + days);
  } // end main
} // end class Mouse3Driver
```

Argument Passing

```
public class Mouse3
{
  private int age = 0;                       // age in days
  private double weight = 1.0;               // weight in grams
  private double percentGrowthRate = 10; // % daily weight gain

  //*************************************************

  public void grow(int days)
  {
    this.age += days;
    while (days > 0)
    {
      this.weight +=
        (.01 * this.percentGrowthRate * this.weight);
      days--;
    }
  } // end grow
} // end class Mouse3
```

Argument Passing

- Java uses the *pass-by-value* mechanism to pass arguments to methods.
- Pass-by-value means that the JVM passes a <u>copy</u> of the argument's value (not the argument itself) to the parameter.
- Thus, if the parameter's value changes within the method, the argument in the calling module is unaffected.
- For example, in the previous two program slides, even though the `days` value within the `grow` method changes, the `main` method's `days` value is unaffected.

Argument Passing

- An argument and its associated parameter often use the same name. For example, we use `days` for the argument in `Mouse3Driver`'s grow method call, and we also use `days` for the parameter in `Mouse3`'s `grow` method heading.
- But be aware that an argument and its associated parameter don't have to use the same name. The only requirement is that an argument and its associated parameter are the same type.
- For example, if `num` is an `int` variable, then this method call successfully passes `num`'s value into the `days` parameter:

```
minnnie.grow(num);
```

- As another example, since 365 is an `int` value, the following method call successfully passes 365 into the `days` parameter:

```
minnie.grow(365);
```

Specialized Methods

- *Accessor* methods -
  - They simply get/access the value of an instance variable.
  - Example:

```
public int getAge()
{
  return this.age;
}
```

- *Mutator* methods -
  - They simply set/mutate the value of an instance variable.
  - Example:

```
public void setPercentGrowthRate(double percentGrowthRate)
{
  this.percentGrowthRate = percentGrowthRate;
} // end setPercentGrowthRate
```

Specialized Methods

- `boolean` methods -
  - They check the truth or falsity of some condition.
  - They always return a `boolean` value.
  - They should normally start with "is".

- For example, here's an `isAdolescent` method that determines whether a `Mouse` object's age is ≤ 100:

```
public boolean isAdolescent()
{
  if (this.age <= 100)
  {
    return true;
  }
  else
  {
    return false;
  }
} // end isAdolescent
```

- Here's how the `isAdolescent` method might be used in `main`:

```
Mouse pinky = new Mouse();
...
if (pinky.isAdolescent() == false)
{
  System.out.println(
    "The Mouse's growth is no longer" +
    " being simulated - too old.");
}
```

Chapter 7
Object-Oriented Programming – Additional Details

- Object Creation - a Detailed Analysis
- Assigning a Reference
- Testing Objects For Equality
- Passing References as Arguments
- Method-Call Chaining
- Overloaded Methods
- Constructors
- Overloaded Constructors

Object Creation - a Detailed Analysis

- Let's start the chapter with a behind-the-scenes detailed look at what happens when a program instantiates an object and stores its address in a referenc
- Code fragment:

  1. `Car car;`
  2. `car = new Car();`
  3. `car.year = 2008;`

reference variable declaration

object instantiaton

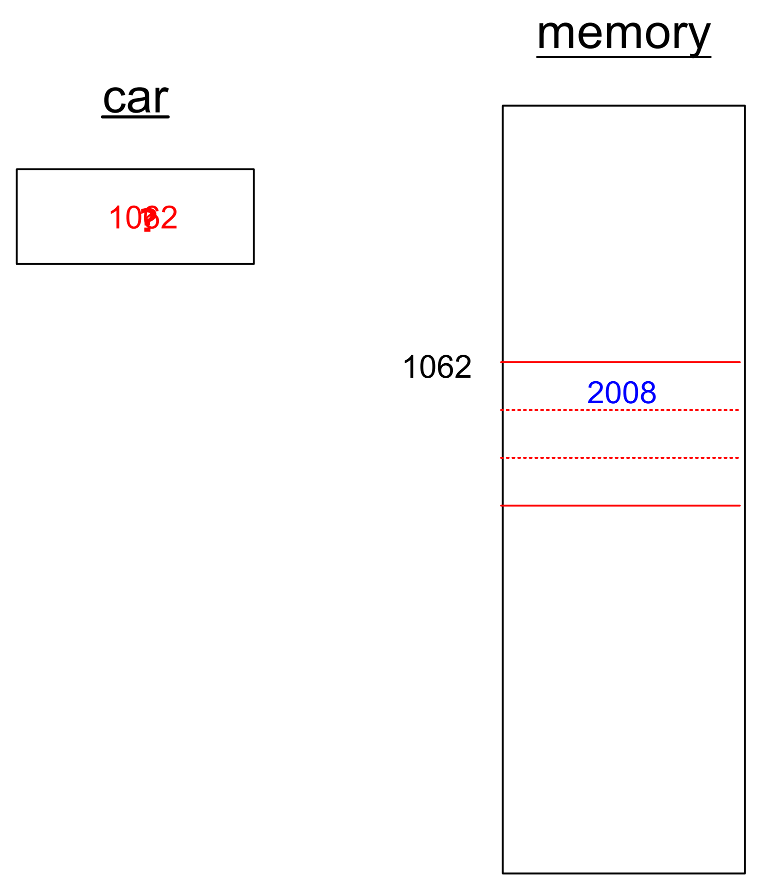
Assign 2008 to `car`'s `year` instance variable

1. Space is allocated in memory for the `car` reference variable. The `car` reference variable will hold the address of an object, but since there's no object created for it yet, it doesn't yet hold a legitimate address.
2. Space is allocated in memory for a new `Car` object. The address of the allocated space is assigned to `car`.
3. The `car` variable's value (the address of a `Car` object) is used to find the `Car` object in memory, and then 2008 can be stored in the `Car` object. Note that for this assignment to work, we're making the simplifying assumption that `year` is a `public` instance variable.

Object Creation - a Detailed Analysis

- **Code fragment:**

```
1. Car car;
2. car = new Car();
3. car.year = 2008;
```



Assigning a Reference

- The result of assigning one reference variable to another is that both reference variables then point to the same object.
- With both reference variables pointing to the same object, if the object is updated by one of the reference variables, then the other reference variable will notice that change when it attempts to access the object.
- That can be disconcerting!

Assigning a Reference

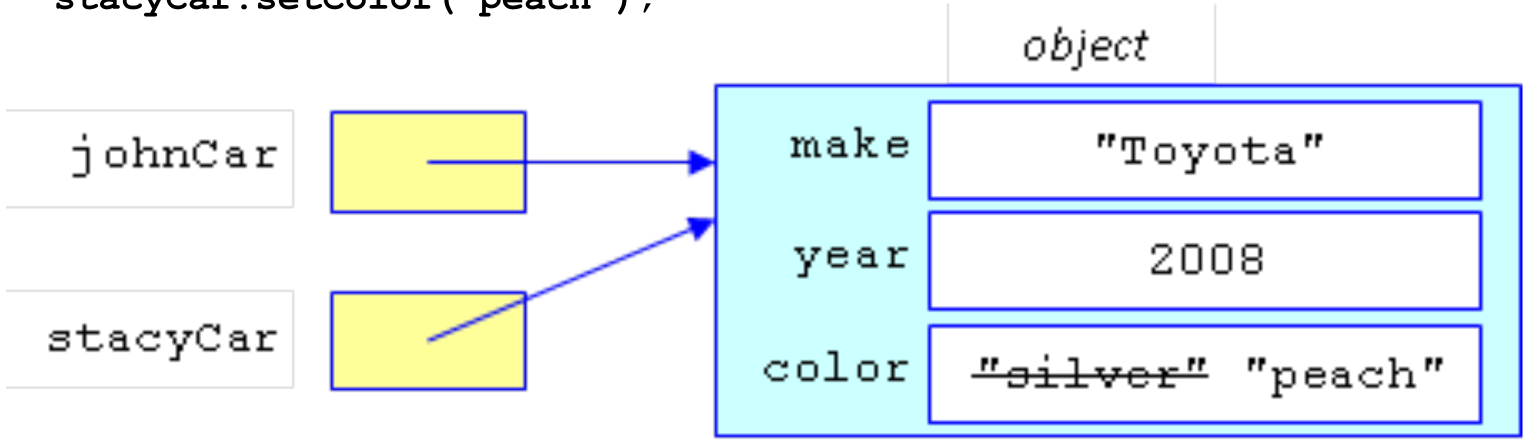
- **Suppose you want to create two `Car` objects that are the same except for their color. Your plan is to create the first car, copy the first car to the second car, and then update the second car's `color` instance variable. Will this code accomplish that?**

```
Car stacyCar;
Car johnCar = new Car();
johnCar.setMake("Toyota");
johnCar.setYear(2008);
johnCar.setColor("silver");
stacyCar = johnCar;
stacyCar.setColor("peach");
```

Assigning a Reference

- **The problem with the previous slide's code is that the `stacyCar = johnCar;` statement causes the two references to point to the same single `Car` object. Thus, `johnCar`'s color becomes "peach" (and that was not intended).**

```
johnCar = new Car();
...
stacyCar = johnCar;
stacyCar.setColor("peach");
```

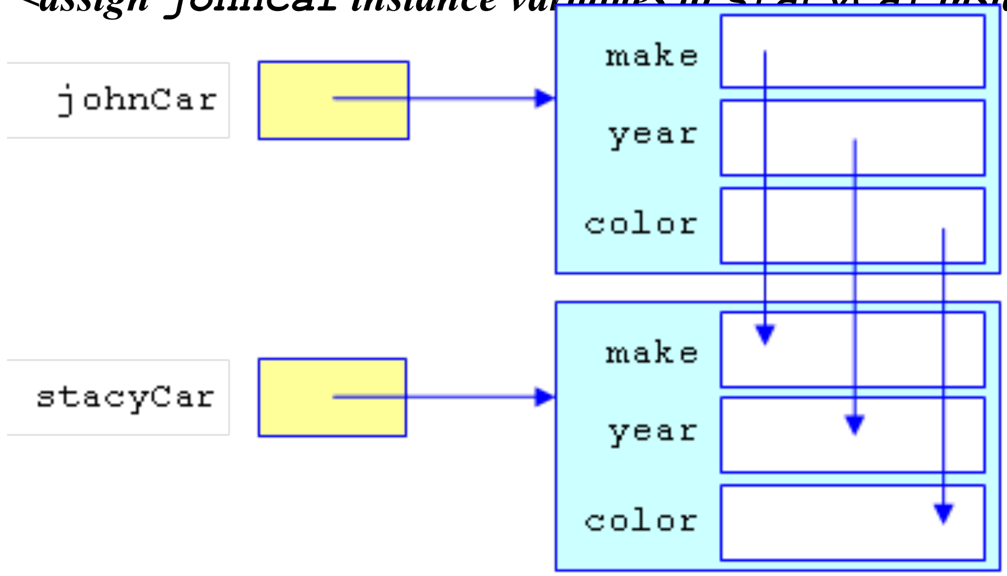


Assigning a Reference

- **If you want to make a copy of a reference variable, you should not assign the reference to another reference. Instead, you should instantiate a new object for the second reference and then assign the two objects' instance variables one at a time.**

```
johnCar = new Car();
stacyCar = new Car();
```

<assign `johnCar` instance variables to `stacyCar` instance variables>



Assigning a Reference

- On the next slide, we make a copy of the `johnCar` reference variable by calling a `makeCopy` method.
- The `makeCopy` method implements the strategy outlined on the previous slide - it instantiates a new object and then copies instance variables into it one at a time. More specifically, the `makeCopy` method:
  - Instantiates a local variable named `car`.
  - Copies the calling object car's instance variables into the local variable `car`'s instance variables.
  - Returns the local variable `car` to the `calling module`.

Assigning a Reference

```
public static void main(String[] args)
{
  Car johnCar = new Car();
  Car stacyCar;

  johnCar.setMake("Toyota");
  johnCar.setYear(2008);
  johnCar.setColor("silver");
  stacyCar = johnCar.makeCopy();
  stacyCar.setColor("peach");
} // end main
```

Assigning a Reference

```
public class Car
{
  private String make;
  private int year;
  private String color;
  ...
  public Car makeCopy()
  {
    Car car = new Car();
    car.make = this.make;
    car.year = this.year;
    car.color = this.color;
    return car;
  } // end makeCarCopy
} // end class Car
```

Testing Objects for Equality

- Using the **==** operator:
  - When comparing two reference variables with ==, you'd probably expect == to return `true` if the data in the two reference variables is the same. Unfortunately, that's not how things work. For example, this prints "different":

```
Car car1 = new Car();
car1.setColor("red");
Car car2 = new Car();
car2.setColor("red");
if (car1 == car2)
{
  System.out.println("the same");
}
else
{
  System.out.println("different");
}
```

The car1 == car2 expression returns `false`. Why?

Testing Objects for Equality

- Using the == operator (continued):
  - The == operator returns `true` if the two reference variables point to the same object; i.e., the two reference variables contain the same address. For example, what does this code fragment print?

```
Car car1 = new Car();
Car car2 = car1;
if (car1 == car2)
{
  System.out.println("the same");
}
else
{
  System.out.println("different");
}
```

Testing Objects for Equality

- Usually, the **==** operator is not good enough. You'll usually want to compare the contents of two objects rather than just whether two reference variables point to the same object.
- To do that, you'll need to have an `equals` method in the object's class definition that compares the contents of the two objects.

Testing Objects for Equality

- Write an `equals` method for a `Car2` class. Use this skeleton:

```
public class Car2
{
  private String make;
  private int year;
  private String color;
  <equals method goes here>
} // end class Car2

public class Car2Driver
{
  public static void main(String[] args)
  {
    Car2 hamoudCar = new Car2();
    Car2 jessicaCar = new Car2();
    ...
    if (hamoudCar.equals(jessicaCar))
    {
      System.out.println("cars have identical features");
    }
```

Passing References as Arguments

- Suppose you pass a reference variable to a method, and inside the method you update the reference variable's instance variables. What happens? …
- Remember that a reference variable holds the address of an object, not the object itself.
- So in passing a reference variable argument to a method, a copy of the object's <u>address</u> (not a copy of the object itself) is passed to the method and stored in the method's parameter.
- Since the parameter and the argument hold the same address value, they point to the same object. Thus, if one of the parameter's instance variables is updated, then the update will simultaneously update the argument's instance variable in the calling module.

Passing References as Arguments

```
public class PersonDriver
{
  public static void main(String[] args)
  {
    Person person1 = new Person();
    Person person2 = new Person();

    person1.setName("Kamal");
    person2.setName("Luis");
    System.out.println(person1.getName()
      + ", " + person2.getName());

    person1.swapPerson(person2);
    System.out.println(person1.getName()
      + ", " + person2.getName());
  } // end main
} // end PersonDriver
```

Aside: Swapping algorithm

- Write a pseudocode fragment that swaps the contents of the x and y variables. More specifically, fill in the swap code below such that the output is "x=8, y=3".

```
x ← 3
y ← 8
<swap code goes here>
print "x=" + x + ", y=" + y
```

Passing References as Arguments

```
public class Person
{
  private String name;

  public void setName(String name)
  {
    this.name = name;
  }

  public String getName()
  {
    return this.name;
  }

  public void swapPerson(Person otherPerson)
  {
    String temp;
    temp = otherPerson.name;
    otherPerson.name = this.name;
    this.name = temp;
  } // end swapPerson
} // end Person
```

Method-Call Chaining

- Up to this point, we've called methods one at a time. In an earlier example, we had a `johnCar` reference variable and we set its `make` and `year` like this:

```
johnCar.setMake("Toyota");
johnCar.setYear(2008);
```

- Let's now discuss how you can chain the two method calls together, like this:

```
johnCar.setMake("Toyota").setYear(2008);
```

- That's called *method-call chaining*. It's when you use a dot to concatenate a method call to the end of another method call.

Method-Call Chaining

```
public class Car3Driver
{
  public static void main(String[] args)
  {
    Car3 car = new Car3();

    car.setMake("Toyota").setYear(2008).printIt();
  } // end main
} // end class Car3Driver
```

a method-call chain

Method-Call Chaining

```
public class Car3
{
  private String make;
  private int year;

  //*******************************************************

  public Car3 setMake(String make)
  {
    this.make = make;
    return this;
  } // end setMake

  public Car3 setYear(int year)
  {
    this.year = year;
    return this;
  } // end setYear

  //*******************************************************

  public void printIt()
  {
    System.out.println(make + ", " + year);
  } // end printIt
} // end class Car3
```

The return type is the same as the class name.

Return the calling object.

Method-Call Chaining

- In `Car3`'s `setMake` and `setYear` methods, note how we enable method-call chaining. In each method definition:
  - The last line in the method body returns the calling object:

    ```
    return this;
    ```
  - The method heading specifies the method's class name for the return type:

    ```
    public Car3 setMake(String make);
    ```

- Method-call chaining is optional. So why bother with it?

Overloaded Methods

- Suppose there's a need to perform the same sort of task on different sets of arguments. For example, suppose you want to find the average for these different sets of arguments:
  - two integers
  - three integers
  - two doubles
- One solution is to write three methods with three different names. Here's how you might call those methods:

```
x = findAverageFor2Ints(20, 8);
y = findAverageFor3Ints(5, -3, 18);
z = findAverageFor2Doubles(1.2, 4.0);
```

- What's wrong with that solution?

Overloaded Methods

- The better solution is to use *overloaded methods*. That's where you have two or more methods with the same name and different parameters (different number of parameters or different types of parameters).
- For the find-the-average example, you could write three overloaded `findAverage` methods and call them like this:

```
x = findAverage(20, 8);
y = findAverage(5, -3, 18);
z = findAverage(1.2, 4.0);
```

Overloaded Methods

```
class Height
{
  double height; // a person's height
  String units;  // unit of measurement (e.g., cm for centimeters)

  public void setHeight(double height)
  {
    this.height = height;
    this.units = "cm";
  }

  public void setHeight(double height, String units)
  {
    this.height = height;
    this.units = units;
  }

  public void print()
  {
    System.out.println(this.height + " " + this.units);
  }
} // end class Height
```

Note that the overloaded `setHeight` methods have different numbers of parameters.

Overloaded Methods

```
public class HeightDriver
{
  public static void main(String[] args)
  {
    Height myHeight = new Height();

    myHeight.setHeight(72.0, "in");
    myHeight.print();
    myHeight.setHeight(180);
    myHeight.print();
  } // end main
} // end class HeightDriver
```

- For each `setHeight` call, which method is called on the previous slide?
- What is the program's output?

Overloaded Methods

- Suppose that you have overloaded methods and you're inside one of the methods. Note that it's OK to call one of the other overloaded methods.
- For example, you can replace the original one-parameter `setHeight` method with the following implementation, which calls the two-parameter `setHeight` method.

```
public void setHeight(double height)
{
  setHeight(height, "cm");
}
```

No need for a reference variable dot prefix here.

Constructors

- Up to this point, we have used mutators to assign values to the instance variables in newly instantiated objects. That works OK, but it requires having and calling one mutator for each instance variable.
- As an alternative, you could use a single method to initialize all of an object's instance variables after you create that object. For example, you could define a single `initCar` method to initialize `Car` objects and use it like this::

```
Car brandonCar = new Car();
brandonCar.initCar("Porsche", 2006, "beige");
```

- The above code fragment uses one statement to allocate space for a new object, and it uses another statement to initialize that object's instance variables. Since the instantiation and initialization of an object is so common, wouldn't it be nice if there were a single statement that could handle both of these operations?

```
Car brandonCar = new Car("Porsche", 2006, "beige");
```

Constructors

- A constructor lets you specify what happens to an object when it is instantiated with `new`.
- A constructor is called automatically when an object is instantiated.
- A constructor's name = the object's class name.
- Don't put a return type at the left of a constructor heading (because constructors never return anything).

Example Car Class with a Constructor

```
public class Car4Driver
{
  public static void main(String[] args)
  {
    Car4 chrisCar = new Car4("Prius", 2008, "blue");
    Car4 goldengCar = new Car4("Volt", 2011, "red");

    System.out.println(chrisCar.getMake());
  } // end main
} // end class Car4Driver
```

constructo r calls

Example Car Class with a Constructor

```
public class Car4
{
  private String make;   // car's make
  private int year;      // car's manufacturing year
  private String color;  // car's primary color

  //*****************************************************

  public Car4(String m, int y, String c)
  {
    this.make = m;
    this.year = y;
    this.color = c;
  } // end constructor

  //**********************************

  public String getMake()
  {
    return this.make;
  } // end getMake
} // end class Car4
```

constructor definition

Style requirement: Put constructors above a class's methods.

Constructors

- Any time you instantiate an object (with `new`), there must be a matching constructor. That is, the number and types of arguments in your constructor call must match the number and types of parameters in a defined constructor.
- Until recently, we've instantiated objects without any explicit constructor. So were those examples wrong?
- The Java compiler automatically provides an empty-bodied zero-parameter default constructor for a class if and only if the class contains no explicitly defined constructors.
- The `Employee` program on the next slide illustrates the use of Java's implicit zero-parameter default constructor.

Will this program compile successfully? Will the next slide’s program compile successfully?

```
import java.util.Scanner;

public class Employee
{
  private String name;

  public void readName()
  {
    Scanner stdIn = new Scanner(System.in);
    System.out.print("Name: ");
    this.name = stdIn.nextLine();
  } // end readName
} // end class Employee


public class EmployeeDriver
{
  public static void main(String[] args)
  {
    Employee emp = new Employee();
    emp.readName();
  } // end main
} // end class EmployeeDriver
```

Will this program compile successfully?

```
import java.util.Scanner;

public class Employee2
{
  private String name;

  public Employee2(String n)
  {
    this.name = n;
  } // end constructor

  public void readName()
  {
    Scanner stdIn = new Scanner(System.in);
    System.out.print("Name: ");
    this.name = stdIn.nextLine();
  } // end readName
} // end class Employee2

public class Employee2Driver
{
  public static void main(String[] args)
  {
    Employee2 waiter = new Employee2("Derrick");
    Employee2 hostess = new Employee2();

    hostess.readName();
  } // end main
} // end class Employee2Driver
```

Overloaded Constructors

- *Constructor overloading* occurs when there are two or more constructors with the same name and different parameters.
- To call an overloaded constructor from another overloaded constructor, use this syntax:

  `this(`*<arguments for target constructor>*`);`
- A `this(`*<arguments-for-target-constructor>*`)` constructor call may appear only in a constructor definition, and it must appear as the very first statement in the constructor definition.
- See the example on the next slide….

Overloaded Constructors

```
public class Fraction
{
  private int numerator;
  private int denominator;
  private double quotient;

  public Fraction(int n)
  {
    this(n, 1);
  }

  public Fraction(int n, int d)
  {
    this.numerator = n;
    this.denominator = d;
    this.quotient =
      (double) this.numerator
      / this.denominator;
  }
```

```
  public void printIt()
  {
    System.out.println(
      this.numerator +
      " / " + this.denominator +
      " = " + this.quotient;
  } // end printIt
} // end Fraction class


public class FractionDriver
{
  public static void main(String[] args)
  {
    Fraction a = new Fraction(3, 4);
    Fraction b = new Fraction(3);
//  Fraction c = new Fraction(); // error
    a.printIt();
    b.printIt();
  } // end main
} // end class FractionDriver
```

Chapter 9 - Classes with Class Members

- Class Variables
- Class Methods
- How to Access Class Members
- When to Use Class Members
- Class Constants
- Example Program Using Class Members

Class Variables

- Based on what you've learned so far, when you're working on an object-oriented program, you should envision separate objects, each with their own set of data and behaviors (instance variables and instance methods, respectively).
- That's a valid picture, but you should be aware that in addition to data and behaviors that are specific to individual objects, you can also have data and behaviors that relate to an entire class. Since they relate to an entire class, such data and behaviors are referred to as *class variables* and *class methods*, respectively.

- For a particular class, each of the class's objects has its own copy of the class's instance variables.
- For a particular class, each of the class's objects shares a single copy of the class's class variables.
- For a mouse growth simulation program, name some appropriate instance variables and some appropriate class variables.

Class Variables

- If you'd like a variable to be shared by all the objects within the variable's class, make it a class variable by using the `static` modifier in its declaration:

  <private or public> `static` *<type> <variable-name>*`;`

- Example:

```
public class Mouse
{
  private static int mouseCount;
  private static double averageLifeSpan;
  ...
```

- Class variables are declared at the top of the class, above all the methods.

Class Variables

- Class variables use the same default values as instance variables:
  - integer types get `0`
  - floating point types get `0.0`
  - boolean types get `false`
  - reference types get `null`
- What are the default values for the class variables in this code fragment?

```
public class Mouse
{
  private static int mouseCount;
  private static double averageLifeSpan;
  private static String researcher;
  private static int simulationDuration = 730;
  ...
```

Initializations are allowed.

Scope

- You can access a class variable from anywhere within its class; i.e., you can access class variables from instance methods as well as from class methods.
- That contrasts with instance variables, which you can access only from instance methods.
- Thus, class variables have broader scope than instance variables. Local variables, on the other hand, have narrower scope than instance variables. They can be accessed only within one particular method.

Scope

- Here is the scope continuum:

local variables — **instance variables** — **class variables**

narrowest scope … **broadest scope**

- Narrower scope equates to more encapsulation, and encapsulation means you are less vulnerable to inappropriate changes.
- Class variables, with their broad scope and lack of encapsulation, can be accessed and updated from many different places, and that makes programs hard to understand and debug. Having broader scope is necessary at times, but in general you should try to avoid broader scope.
- Thus, you should prefer local variables over instance variables and instance variables over class variables.

Class Methods

- If you have a method that accesses class variables and not instance variables, then you should declare the method to be a *class method*. To do so, add `static` to the method's heading like this:

  `<private-or-public> static <return-type> <method-name>(<parameters>)`

- Example:

```
public class Mouse
{
  private static int mouseCount;
  private static double averageLifeSpan;

  public static void printMouseCount()
  {
    System.out.println("Total mice = " +
      Mouse.mouseCount);
  }
}
```

To access a class variable, prefix it with *<class-name>* dot.

How to Access Class Members

- **Normally, to access a class member (a class variable or a class method), prefix it with** *<class-name>* **dot.**
- **Shortcut syntax for a class member:**
  - **In accessing a class member, you may omit the** *<class-name>* **dot prefix if the class member is in the same class as where you're trying to access it from. For example:**

```
public class Mouse
{
  private static int mouseCount;
  private static void printMouseCount()
  {
    System.out.println("Total mice = " + Mouse.mouseCount);
  }
  public static void main(String[] args)
  {
    Mouse.printMouseCount();
  }
} // end Mouse class
```

OK to access `printMouseCount` **without** *<class-name>* **dot.**

OK to access `mouseCount` **without** *<class-name>* **dot.**

When to Use Class Methods

- You should make a method a class method:
  - If you have a method that uses class variables and/or calls class methods, then it's a good candidate for being a class method. Warning: If in addition to accessing class members, the method also accesses instance members, then the method must be an instance method, not a class method.
  - If you might need to call a method even when there are no objects from the method's class, then you should make it a class method.
  - The `main` method has to be a class method. If a `main` method uses helper methods that don't involve instance members, then the helper methods should be class methods as well.

Class Constants

- Sometimes, you'll want a class variable to be fixed/constant. That type of "variable" is called a *class constant*.
- To make a class constant, declare a variable with the `static` and `final` modifiers like this:

  <private or public> `static final` *<type> <variable-name>* `=` *<initial value>*`;`

- Note that class constants should be assigned as part of an initialization statement. If you attempt to assign a value to a class constant later on, that generates a compilation error.
- As with most variables, class constants usually use the `private` modifier. However, if the constant is so important that more than one class needs to use it, then you should use the `public` modifier.

Class Constants

- In the Human class below, we make NORMAL_TEMP a class constant (with the static and final modifiers) because all Human objects have the same normal temperature of 98.6° Fahrenheit.

class constant

```
public class Human
{
  private static final double NORMAL_TEMP = 98.6;
  ...
  public boolean isHealthy()
  {
    return Math.abs(currentTemp - NORMAL_TEMP) < 1;
  } // end isHealthy

  public void diagnose()
  {
    if ((currentTemp - NORMAL_TEMP) > 5)
    {
      System.out.println("Go to the emergency room now!");
      ...

} // end class Human
```

Class Constants

- Prior to this chapter, you used named constants that were declared locally within a method. For example:

```
public void calculateSignalDelay(double cableDistance)
{
  final double SPEED_OF_LIGHT = 299792458.0;
  double propagationDelay;  // time for electron to travel
                            // length of cable
  ...
  propagationDelay = cableDistance / SPEED_OF_LIGHT;
  ...
}
```

- So when should you use a local variable named constant and when should you use a class constant?
  - Use a local variable named constant if the constant is needed only within one method.
  - Use a class constant if the constant is a property of the collection of all the objects in the class or of the class in general.

Example Program Using Class Members

```
/*************************************************************
* PennyJar.java
* Dean & Dean
*
* This class counts pennies for individual penny jars and for
* all penny jars combined.
*************************************************************/

public class PennyJar
{
  public static final int GOAL = 10000;
  private static int allPennies = 0;
  private int pennies = 0;

  //**********************************************************

  public int getPennies()
  {
    return this.pennies;
  }

  //**********************************************************

  public void addPenny()
  {
    System.out.println("Clink!");
    this.pennies++;
    PennyJar.allPennies++;

    if (PennyJar.allPennies >= PennyJar.GOAL)
    {
      System.out.println("Time to spend!");
    }
  } // end addPenny
```

Example Program Using Class Members

```
  //**********************************************************

  public static int getAllPennies()
  {
    return PennyJar.allPennies;
  }
} // end class PennyJar

/**********************************************************
* PennyJarDriver.java
* Dean & Dean
*
* This class drives the PennyJar class.
**********************************************************/

public class PennyJarDriver
{
  public static void main(String[] args)
  {
    PennyJar pennyJar1 = new PennyJar();
    PennyJar pennyJar2 = new PennyJar();

    pennyJar1.addPenny();
    pennyJar1.addPenny();
    pennyJar2.addPenny();
    System.out.println(pennyJar1.getPennies());
    System.out.println(PennyJar.getAllPennies());
  } // end main
} // end class PennyJarDriver
```

Chapter 12 – Aggregation, Composition, and Inheritance

- Composition
- Aggregation
- UML Class Diagram for Composition and Aggregation
- Car Dealership Program
- Inheritance Overview
- Inheritance Example - People in a Department Store
- Inheritance Terminology
- UML Class Diagrams for Inheritance Hierarchies
- Benefits of Inheritance
- Inheritance For a Superclass's `private` Instance Variables
- Using `super` to Call Superclass Constructor
- Calling a Superclass's Method from Within a Subclass
- Default Call to Superclass Constructor
- Method Overriding
- The `final` Access Modifier
- Aggregation , Composition, and Inheritance Compared
- Aggregation, Composition, and Inheritance Combined
- Card Game Program

Composition

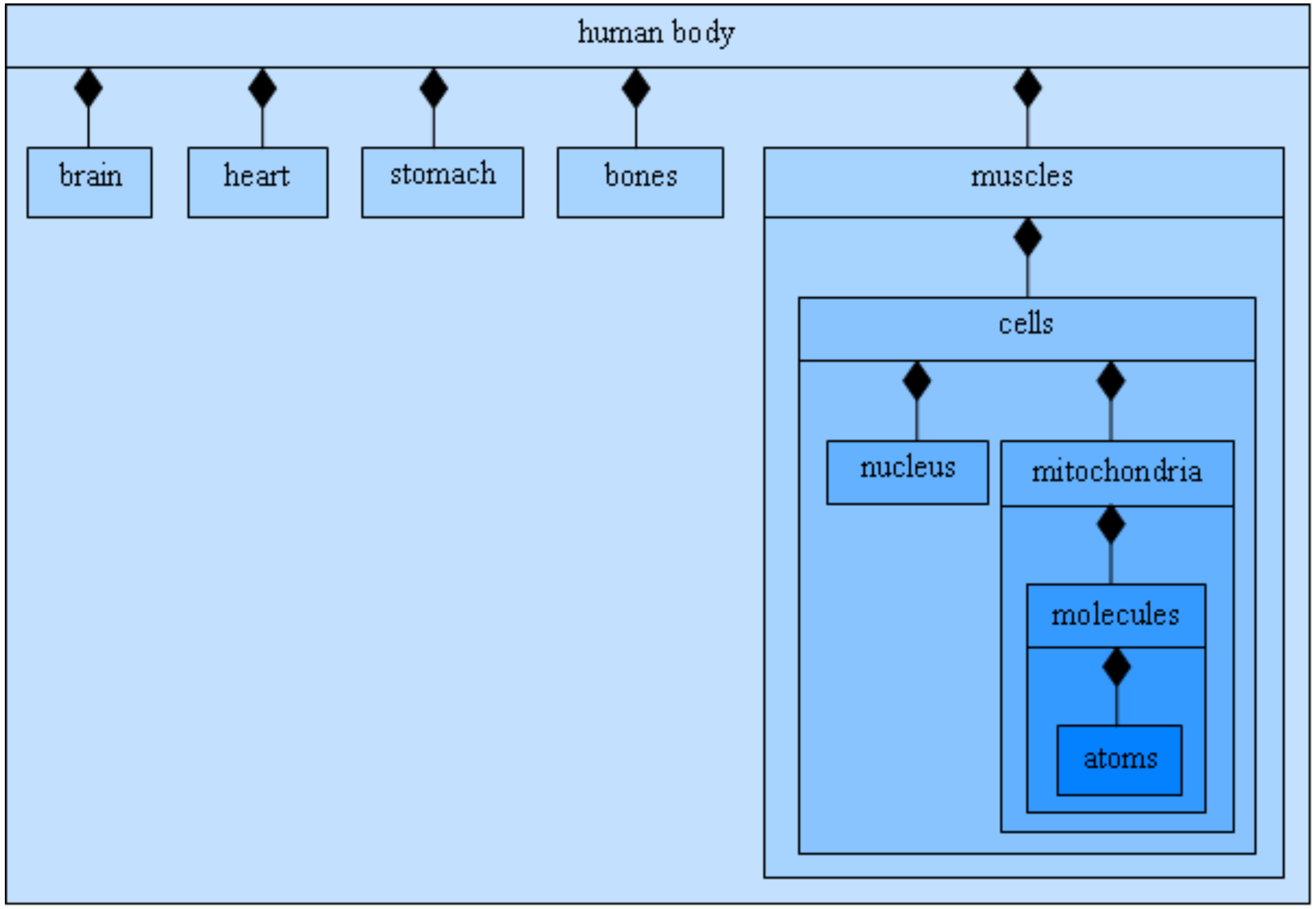
- Prior to this chapter, all of our objects have been relatively simple, so we've been able to describe each object with just a single class.
- But for an object that's more complex, you should consider breaking up the object into its constituent parts and defining one class as the whole and other classes as parts of the whole. When the whole class is the exclusive owner of the parts classes, then that class organization is called a *composition*.

Composition

- The concept of composition is not new; that's what we do to describe complex objects in the real world:
  - Every living creature and most manufactured products are made up of parts. Often, each part is a subsystem that is itself made up of its own set of subparts. Together, the whole system forms a composition hierarchy.
- Note the human body composition hierarchy on the next slide.
- Remember that with a composition relationship, a component part is limited to just one owner at a time. For example, a heart can be in only one body at a time.

Composition

- **A partial composition hierarchy for the human body:**



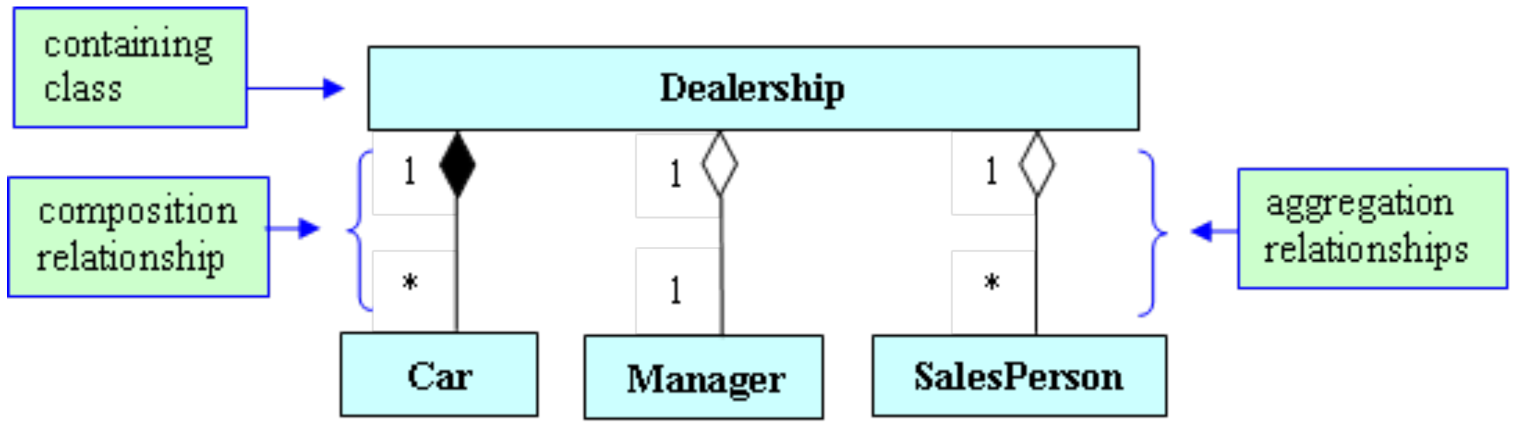
Aggregation

- In a composition hierarchy, the relationship between a containing class and one of its part classes is known as a *has-a* relationship. For example, each human body has a brain and has a heart.
- There's another has-a relationship, called *aggregation*, which is a weaker form of composition. With aggregation, one class is the whole and other classes are parts of the whole (as with composition), but there is no additional constraint that requires parts to be exclusively owned by the whole.
- An aggregation example where the parts are not exclusively owned by the whole –
  - You can implement a school as an aggregation by creating a whole class for the school and part classes for the different types of people who work and study at the school.
  - The people aren't exclusively owned by the school because a person can be part of more than one aggregation.
  - For example, a person can attend classes at two different schools and be part of two school aggregations. The same person might even be part of a third aggregation, of a different type, like a household aggregation.

Car Dealership Program

- Suppose you're trying to model a car dealership with a computer program. Since the car dealership is made from several distinct non-trivial parts, it's a good candidate for being implemented with composition and aggregation relationships.
- The car dealership's "non-trivial parts" are a group of cars, a sales manager, and a group of sales people.
- In implementing the program, define four classes:
  - The `Car`, `Manager`, and `SalesPerson` classes implement the car dealership's non-trivial parts.
  - The `Dealership` class contains the three parts classes.
- For each of the three class relationships, `Dealership`-`Car`, `Dealership`-`SalesPerson`, and `Dealership`-`Manager`, is the relationship composition or aggregation?

UML Class Diagram for Composition and Aggregation



- **Universal Modeling Language (UML) class diagrams show the relationships between a program's classes:**
  - **A solid line between two classes represents an *association* – a relationship between classes.**
  - **On an *association line*, a solid diamond indicates a composition relationship, and a hollow diamond indicates an aggregation relationship. The diamond goes next to the container class.**
  - **The labels on the association lines are called *multiplicity values*. They indicate the number of object instances for each of the two connected classes.**
  - **The * multiplicity value represents any size number, zero through infinity.**

Car Dealership Program

- To implement a program that uses aggregation and composition:
  - Define one class for the whole and define separate classes for each of the parts.
  - For a class that contains another class, declare an instance variable inside the containing class such that the instance variable holds a reference to one or more of the contained class's objects.
  - Typically, for association lines with * multiplicity values, use an `ArrayList` to implement the instance variable associated with the asterisked class.

  - If two classes have an aggregation relationship with non-exclusive ownership, then store the contained class's object in an instance variable in the containing class, but also store it in another variable outside of the containing class, so the object can be added to another aggregation and have two different "owners."
  - If two classes have a composition relationship, then store the contained class's object in an instance variable in the containing class, but do not store it elsewhere. That way, the object can have only one "owner."

Car Dealership Program

```
/*************************************************************
* Dealership.java
* Dean & Dean
*
* This represents an auto retail sales organization.
*************************************************************/

import java.util.ArrayList;

public class Dealership
{
  private String company;
  private Manager manager;
  private ArrayList<SalesPerson> people = new ArrayList<SalesPerson>();
  private ArrayList<Car> cars = new ArrayList<Car>();

  //***********************************************************

  public Dealership(String company, Manager manager)
  {
    this.company = company;
    this.manager = manager;
  }
```

Car Dealership Program

```
  //***********************************************************

  public void addCar(Car car)
  {
    cars.add(car);
  }

  public void addPerson(SalesPerson person)
  {
    people.add(person);
  }

  //***********************************************************

  public void printStatus()
  {
    System.out.println(company + "\t" + manager.getName());
    for (SalesPerson person : people)
      System.out.println(person.getName());
    for (Car car : cars)
      System.out.println(car.getMake());
  } // end printStatus
} // end Dealership class
```

Car Dealership Program

```
/*************************************************************
* Car.java
* Dean & Dean
*
* This class implements a car.
*************************************************************/

public class Car
{
  private String make;

  //***********************************************************

  public Car(String make)
  {
    this.make = make;
  }

  //***********************************************************

  public String getMake()
  {
    return make;
  }
} // end Car class
```

Car Dealership Program

```
/*************************************************************
* Manager.java
* Dean & Dean
*
* This class implements a car dealership sales manager.
*************************************************************/

public class Manager
{
  private String name;

  //**********************************************************

  public Manager(String name)
  {
    this.name = name;
  }

  //**********************************************************

  public String getName()
  {
    return name;
  }
} // end Manager class
```

Car Dealership Program

```
/*************************************************************
* SalesPerson.java
* Dean & Dean
*
* This class implements a car sales person
*************************************************************/

public class SalesPerson
{
  private String name;
  private double sales = 0.0; // sales to date

  //**********************************************************

  public SalesPerson(String name)
  {
    this.name = name;
  }

  //**********************************************************

  public String getName()
  {
    return name;
  }
} // end SalesPerson class
```

Car Dealership Program

```
/*************************************************************
* DealershipDriver.java
* Dean & Dean
*
* This class demonstrates car dealership composition.
*************************************************************/

public class DealershipDriver
{
  public static void main(String[] args)
  {
    Manager ahmed = new Manager("Ahmed Abdi");
    SalesPerson ash = new SalesPerson("Ash Lawrence");
    SalesPerson jeffrey = new SalesPerson("Jeffrey Leung");
    Dealership dealership = new Dealership("OK Used Cars", ahmed);

    dealership.addPerson(ash);
    dealership.addPerson(jeffrey);
    dealership.addCar(new Car("GMC"));
    dealership.addCar(new Car("Yugo"));
    dealership.addCar(new Car("Dodge"));
    dealership.printStatus();
  } // end main
} // end DealershipDriver class
```

Inheritance Overview

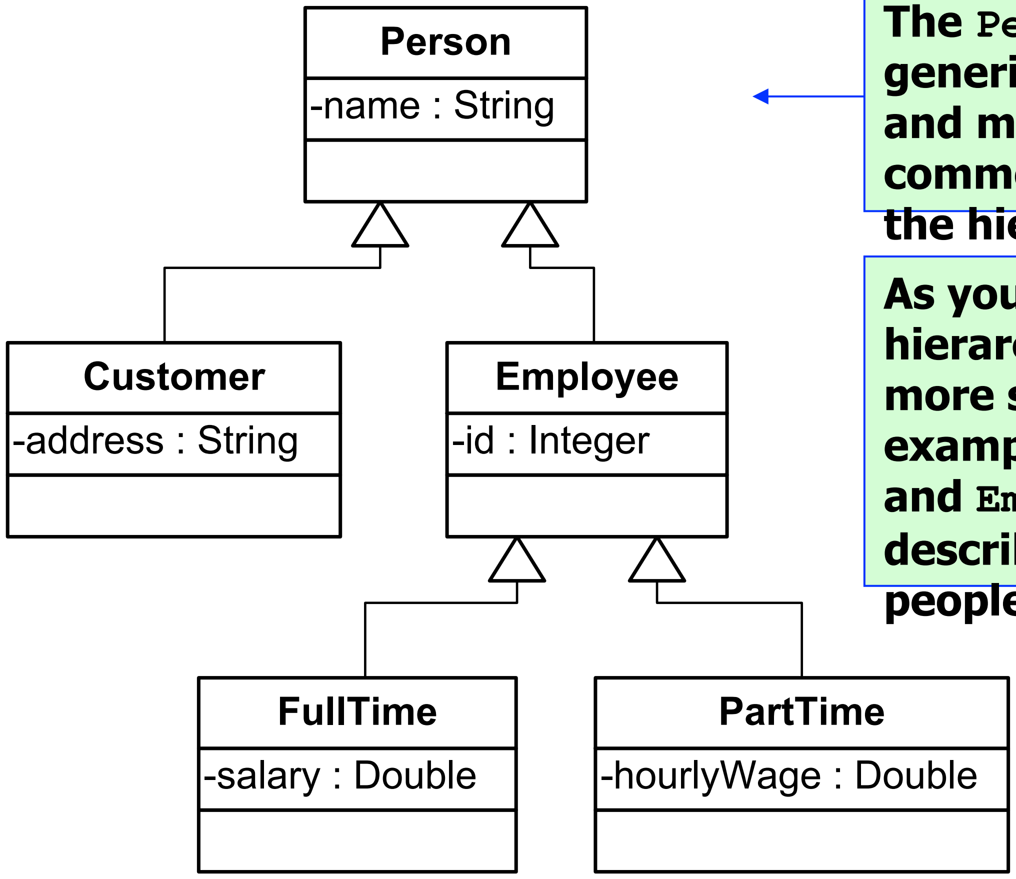
- There are different ways that classes can be related. We've covered aggregation and composition, where one class is the whole and other classes are parts of the whole. *Inheritance* is another type of class relationship….
- Suppose you're in charge of designing cars for a car manufacturer:
  - You could create independent design blueprints for the manufacturer's five car models, but to save time, it's better to first make a master blueprint that describes features that are common to all the models.
  - Then make additional blueprints, one for each model. The additional blueprints describe features that are specific to the individual models.

Inheritance Overview

- Creating a more specific blueprint that's based on an existing master blueprint is analogous to the concept of *inheritance*, in which a new class is derived from an existing class.
- It's called inheritance because the new class inherits/ borrows all the features (data and methods) of the existing class.
- Inheritance is a very important feature of Java (and all OOP languages) because it allows programmers to reuse existing software. More specifically, the existing class is used by all of the classes that are derived from it.

Inheritance Example - People in a Department Store

- Here's a UML class diagram for an inheritance hierarchy that keeps track of people in a department store:



The `Person` class is generic - it contains data and methods that are common to all classes in the hierarchy.

As you go down the hierarchy, the classes get more specific. For example, the `Customer` and `Employee` classes describe specific types of people in the store.

Inheritance Terminology

- Within an inheritance hierarchy, pairs of classes are linked together. For each pair of linked classes, the more generic class is called the *superclass* and the more specific class is called the *subclass*.
- We say that subclasses are *derived* from superclasses. That makes sense when you realize that subclasses *inherit* all of the superclass's data and methods.
- Unfortunately, the terms superclass and subclass can be misleading. The "super" in superclass could imply that superclasses have more capability and the "sub" in subclass could imply that subclasses have less capability. Actually, it's the other way around - subclasses have more capability. Subclasses can do everything that superclasses can do, plus more.
- We'll stick with the terms superclass and subclass since those are the formal terms used by Oracle, but be aware of this alternative terminology:
  - Programmers often use the terms *parent class* or *base class* when referring to a superclass.
  - Programmers often use the terms *child class* or *derived class* when referring to a subclass.

UML Class Diagrams for Inheritance Hierarchies

- Usually, UML class diagrams show superclasses above subclasses. That's a common practice, but not a requirement. The following is a requirement….
- UML class diagrams use an arrow for inheritance relationships, with a hollow arrowhead pointing to the superclass.
- Warning:
  - The direction of arrows in UML class diagrams is opposite to the direction in which inheritance flows. In the previous slide, the `Customer` class inherits the `name` variable from the `Person` class. And yet the arrow does not go from `Person` to `Customer`; it goes from `Customer` to `Person`. That's because the arrow points to the superclass, and `Person` is the superclass.
- UML class diagram review:
  - What are the class boxes' minus signs for?
  - What are the class boxes' third compartments for?

Benefits of Inheritance

- Benefits of inheritance:
  - It helps with code reusability -
    - A superclass's code can be used for multiple subclasses. That eliminates code redundancy and makes debugging and upgrading easier.
    - A programmer can use an existing class to easily create a new subclass (no need to "reinvent the wheel.")
  - Smaller modules (because classes are split into superclasses and subclasses) -
    - That makes debugging and upgrading easier.

`Person-Employee` Example

- Implement a *`Person`* superclass with an *`Employee`* subclass.
- The `Person` class should:
  - Declare a `name` instance variable.
  - Define appropriate constructors.
  - Define a `getName` accessor method.
- The `Employee` class should:
  - Inherit `Person`'s members.
  - Declare an `id` instance variable.
  - Define appropriate constructors.
  - Define a `display` method.

Person-Employee Example

```
public class Person
{
  private String name = "";

  //************************************************************

  public Person()
  { }

  public Person(String name)
  {
    this.name = name;
  }

  //************************************************************

  public String getName()
  {
    return this.name;
  }
} // end Person class
```

Person-Employee Example

```
public class Employee extends Person
{
  private int id = 0;

  //**********************************************************

  public Employee()
  { }

  public Employee(String name, int id)
  {


  }

  //**********************************************************

  public void display()
  {


  }
} // end Employee class
```

Inheritance for a superclass's `private` Instance Variables

- Since `name` is a `private` instance variable in the `Person` superclass, the `Employee` class's methods and constructors cannot access `name` directly (that's the same interpretation of `private` that we've always had).
- So how can `Employee` methods and constructors access `name`?
- By using the `Person` class's `public` methods and constructors!
- Aside - even though a superclass's `private` instance variables are not directly accessible from a subclass, `private` instance variables are still considered to be "inherited" by the subclass. So for this example, each `Employee` object does indeed contain a `name` instance variable.

Using `super` to Call Superclass Constructor

- A constructor may call a constructor in its superclass by using this syntax:

```
super(<arguments>);
```

- A constructor call is allowed only if it's the very first line in the constructor's body. This rule is applicable for constructor calls to constructors in the same class (using `this`) and also for constructor calls to constructors in the superclass (using `super`).
- What happens if you have a `super` constructor call and then a `this` constructor call as the first two lines in a constructor?

Calling a Superclass's Method from Within a Subclass

- As you may recall, in an instance method, if you call a method that's in the same class as the class you're currently in, the reference variable dot prefix is unnecessary.
- Likewise, in an instance method, if you call a method that's in the superclass of the class you're currently in, the reference variable dot prefix is unnecessary.
- Thus, in the `Employee` class's `display` method, call the `Person` class's `getName` method like this:

```
System.out.println("name: " + getName());
```

Default Call to Superclass Constructor

- The Java designers at Oracle are fond of calling superclass constructors since that promotes software reuse.
- If you write a subclass constructor and don't include a call to another constructor (with `this` or with `super`), the Java compiler will sneak in and insert a superclass zero-parameter constructor call by default.
- Thus, although we showed nothing in Employee's zero-parameter constructor, the Java compiler automatically inserts `super();` in it for us. So these two constructors are functionally equivalent:

```
public Employee()
{ }
```

```
public Employee()
{
  super();
}
```

Method Overriding

- *Method overriding* is when a subclass has a method with the same name and the same parameter types as a method in its superclass.
- If a subclass contains an overriding method:
  - By default, an object of the subclass will use the subclass's overriding method (and not the superclass's overridden method).
  - Sometimes, an object of the subclass may need to call the superclass's overridden method. To do that, preface the method call with "super." (don't forget the dot).
- If a subclass and a superclass have methods with the same name, same parameter types, and different return types, that generates a compilation error.

FullTime Class

- Complete the implementation of the `FullTime` class below. In particular, provide a 3-parameter constructor and a `display` method.

```
public class FullTime extends Employee
{
  private double salary = 0.0;

  public FullTime()
  { }







  public static void main(String[] args)
  {
    FullTime fullTimer = new FullTime("Alan O'Brien", 5733, 80000);
    fullTimer.display();
    System.out.println(fullTimer.getName());
  }
} // end FullTime class
```

FullTime Class

- From the previous slide's main method, note this getName call:

```
System.out.println(fullTimer.getName());
```

- fullTimer is a FullTime reference variable, so you would expect it to call methods defined in the FullTime class.
- But fullTimer calls getName, which is defined in the Person class, not the FullTime class. So how can fullTimer call the getName method?
- The Employee class inherits the getName method from the Person class and the FullTime class inherits the inherited getName method from the Employee class.

The `final` Access Modifier

- If you want to specify that a method definition cannot be overridden with a new definition in a subclass, use the `final` modifier in the method heading. For example:

```
public class Person
{
   ...
   public final String getName()
   {
      ...

public class Employee extends Person
{
   ...
   public String getName()
   {
      ...
```

compilation error

The `final` Access Modifier

- Why would you ever want to use `final`?
  - If you think that your method is perfect, then you might want to use `final` to prevent anyone else from overriding it in the future.
  - `final` methods might run faster since the compiler can generate more efficient code for them (because there's no need to prepare for the possibility of inheritance).

- If you want to specify that a class cannot have any subclasses, use the `final` access modifer in the class heading. For example:

```
public final class FullTime
{
   ...

public class FullTimeNightShift extends FullTime
{
   ...
```

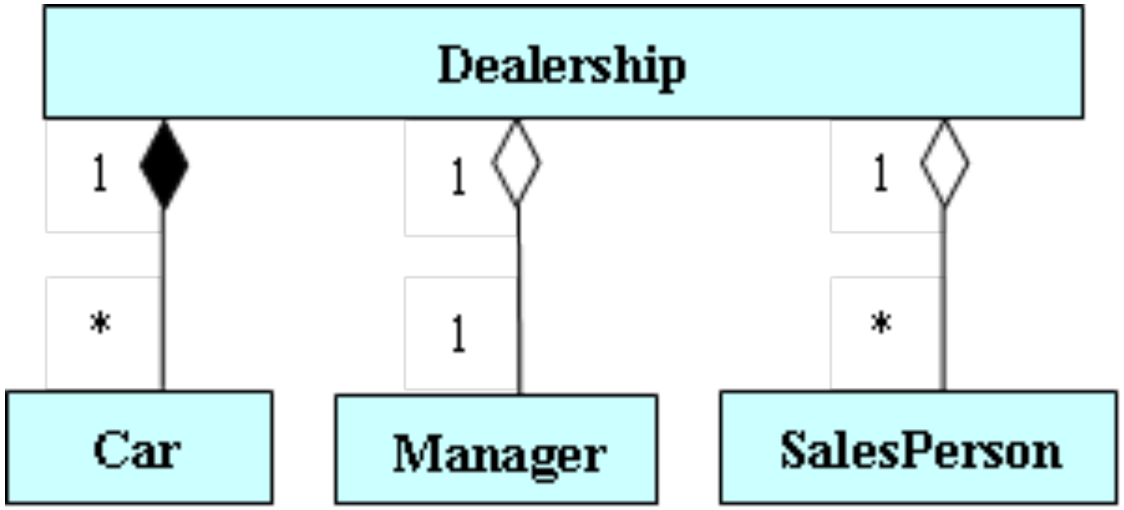
compilation error

Aggregation, Composition, and Inheritance Compared

- We've covered two basic types of class relationships:
  - Aggregation and composition relationships are when one class is a whole and other classes are parts of that whole.
  - An inheritance relationship is when one class is a more detailed version of another class. The more detailed class is a subclass, and the other class is a superclass. The subclass inherits the superclass's members (variables and methods).
- We call aggregation and composition relationships "has a" relationships because one class, the container class, <u>has a</u> component class inside of it.
- We call an inheritance relationship an "is a" relationship because one class, a subclass, <u>is a</u> more detailed version of another class.

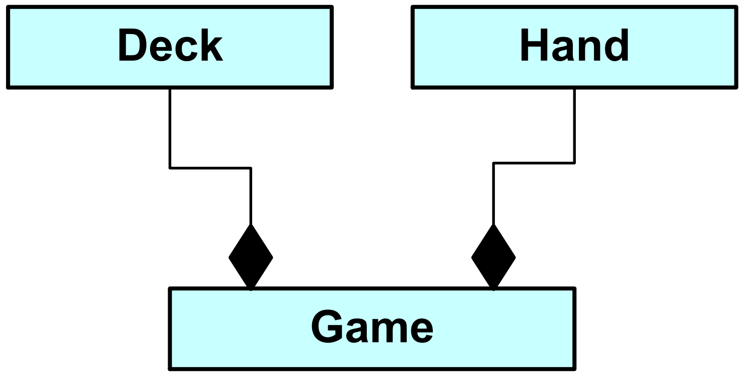
Aggregation, Composition, and Inheritance Combined

- In the real world, it's fairly common to have aggregation, composition, and inheritance relationships in the same program.
- For example, what sort of inheritance relationship could/should be added to our earlier `Dealership` program, shown below?



Card Game Program

- Provide a class diagram for a card game program:
  - Assume it's a game like war or gin rummy where you have a deck of cards and two players.
  - Decide on appropriate classes. For each class, draw a UML-notation three-partition rectangle and put the class name in the top partition.
  - Look for composition relationships between classes. For each pair of classes related by composition, draw a composition connecting line with a diamond next to the containing class. For example, the left composition association line below is for `Game`, the containing class, and `Deck`, the contained class.



Card Game Program

- Provide a class diagram for a card-game program (continued):
  - For each class, decide on appropriate instance variables and put them in the middle partition of the class's rectangle.
  - For each class, decide on appropriate public methods and put their declarations in the bottom partition of the class's rectangle.
  - Look for common instance variables and methods. If two or more classes contain a set of common instance variables and/or methods, implement a superclass and move the common entities to the superclass. For each subclass/ superclass pair, draw an arrow from the subclass to the superclass to indicate an inheritance relationship.

Card Game Program

inheritance implementation

```
public class Deck extends GroupOfCards
{
  public static final int
    TOTAL_CARDS = 52;

  public Deck()
  {
    for (int i=0; i<TOTAL_CARDS; i++)
    {
      addCard(
        new Card((2 + i%13), i/13));
    }
  } // end constructor
  ...
} // end class Deck
```

composition implementation

```
public class Deck
{
  public static final int
    TOTAL_CARDS = 52;
  GroupOfCards groupOfCards =
    new GroupOfCards();

  public Deck()
  {
    for (int i=0; i<TOTAL_CARDS; i++)
    {
      groupOfCards.addCard(
        new Card((2 + i%13), i/13));
    }
  } // end constructor
  ...
} // end class Deck
```

Card Game Program

- Here's a `main` method for the card game program:

```
public static void main(String[] args)
{
  Scanner stdIn = new Scanner(System.in);
  String again;
  Game game;

  do
  {
    game = new Game();
    game.playAGame();
    System.out.print("Play another game (y/n)?: ");
    again = stdIn.nextLine();
  } while (again.equals("y"));
} // end main
```

Card Game Program

- Here's a `playAGame` method for the `Game` class:

```
public void playAGame()
{
  Card card;
  deck.shuffle();

  while (deck.getCurrentSize() > 0)
  {
    card = deck.dealCard();
    player1.addCard(card);
    card = deck.dealCard();
    player2.addCard(card);
  }
  ...
} // end playAGame
```

Chapter 13 - Inheritance and Polymorphism

- The `Object` Class
- The `equals` Method
- The `toString` Method
- Polymorphism
- Dynamic Binding
- Compilation Details
- Polymorphism with Arrays
- Abstract Methods And Classes

The `Object` Class

- The `Object` class is a superclass for all other classes.
- When declaring your own classes, you don't have to specify the `Object` class as a superclass - it's automatically a superclass.

- We're covering just two of `Object`'s methods. The `equals` and `toString` methods are the most important `Object` methods....

The `equals` Method

- For a class that doesn't have its own `equals` method, if an object from that class calls the `equals` method, it inherits and uses the `Object` class's `equals` method.
- The `Object` class's `equals` method returns `true` if the two reference variables that are being compared point to the same object; that is, if the two reference variables contain the same address.

The `equals` Method

- Assuming that the `Car` class does not have its own `equals` method, what does this code fragment print?

```
Car car1 = new Car("Honda");
Car car2 = car1;
if ((car1.equals(car2) && (car1 == car2))
{
  System.out.println("cars are equal - first time");
}
car2 = new Car("Honda");
if ((car1.equals(car2) || (car1 == car2))
{
  System.out.println("cars are equal - second time");
}
```

- Aside: the == operator works the same as the `Object` class's `equals` method; == returns `true` if the two reference variables point to the same object.

The `equals` Method

- Usually, the `Object` class's `equals` method is not good enough. You'll usually want to compare the <u>contents</u> of two objects rather than just whether two reference variables point to the same object.
- To do that, you'll need to have an `equals` method in the object's class definition that compares the contents of the two objects.

Defining Your Own `equals` Method

- Write an `equals` method for a `Car` class. Use this skeleton:

```
public class Car
{
  private String make;
  private int year;
  private String color;
```

  <equals method goes here>

```
} // end class Car

public class CarDriver
{
  public static void main(String[] args)
  {
    Car[] cars = new Car[100];
    cars[0] = new Car("Chevrolet", 2010, "black");
    if (cars[0].equals(cars[1]))
    {
       System.out.println("cars have identical features");
    }
    ...
```

The `equals` Method

- Note that `equals` methods are built into lots of Java's API classes.
- For example, the `String` class and the wrapper classes implement their own `equals` methods.
- As you'd expect, those `equals` methods test whether the contents of the two compared objects are the same (not whether the addresses of the two compared objects are the same).
- What does this code fragment print?

```
String s1 = "hello", s2 = "he";
s2 += "llo";
if (s1 == s2)
{
  System.out.println("\"==\" works");
}
if (s1.equals(s2))
{
  System.out.println("\"equals\" works");
}
```

The `toString` Method

- The `Object` class's `toString` method returns a string that's a concatenation of the calling object's class name, an @ sign, and a sequence of digits and letters (called a *hashcode*).
- Consider this code fragment:

```
Object obj = new Object();
System.out.println(obj.toString());
Car car = new Car();
System.out.println(car.toString());
```

- Here's the output:

```
java.lang.Object@601bb1
Car@1ba34f2
```

The `Object` class is in the `java.lang` package.

hashcode

- If a class is stored in a package, `toString` prefixes the class name with the class's package.

The `toString` Method

- Retrieving the class name, an @ sign, and a hashcode is usually worthless, so you'll almost always want to avoid calling the `Object` class's `toString` method and instead call an overriding `toString` method.
- In general, `toString` methods should return a string that describes the calling object's contents.
- You'll find lots of overriding `toString` methods in the Java API classes.
  - For example, the `Date` class's `toString` method returns a `Date` object's month, day, year, hour, and second values as a single concatenated string.
- Since retrieving the contents of an object is such a common need, you should get in the habit of providing a `toString` method for most of your programmer-defined classes.
  - Typically, your `toString` methods should simply concatenate the calling object's stored data and return the resulting string.
  - Note that `toString` methods should <u>not</u> print the concatenated string value; they should just <u>return</u> it!!!

The `toString` Method

- Write a `toString` method for a `Car` class. Use this skeleton:

```
public class Car
{
  private String make;
  private int year;
  private String color;
  ...
  <toString method goes here>
} // end class Car

public class CarDriver
{
  public static void main(String[] args)
  {
    Car car = new Car("Honda", 1998, "silver");
    System.out.println(car);
    ...
```

The `toString` Method

- The `toString` method is automatically called when a reference variable is an argument in a `System.out print`, `println`, or `printf` call. For example:

```
System.out.println(car);
```

- The `toString` method is automatically called when a reference variable is concatenated (+ operator) to a string. For example:

```
String carInfo = "Car data:\n" + car;
```

- Note that you can also call an object's `toString` method using the standard method-call syntax. For example:

```
car.toString();
```

The `toString` Method

- Write a `toString` method for a `Counter` class. Use this skeleton:

```
public class Counter
{
  private int count;
  ...
  <toString method goes here>
} // end class Counter

public class CounterDriver
{
  public static void main(String[] args)
  {
    Counter counter = new Counter(100);
    String message = "Current count = " + counter;
    ...
```

Wrapper Classes' `toString` Methods

- All the primitive wrapper classes have `toString` methods that return a string representation of the given primitive value. For example:

```
Integer.toString(22)        : evaluates to string "22"
Double.toString(123.45)     : evaluates to string "123.45"
```

Polymorphism

- Polymorphism is when different types of objects respond differently to the same method call.
- To implement polymorphic behavior, declare a general type of reference variable that is able to refer to objects of different types.
- To declare a "general type of reference variable," use a superclass. Later, we'll use a programmer-defined superclass. For now, we'll keep things simple and use the predefined `Object` superclass.
- In the following `Pets` program, note how `obj` is declared to be an `Object` and note how the `obj.toString()` method call exhibits polymorphic behavior:
  - If `obj` contains a `Dog` object, `toString` returns "Woof! Woof!"
  - If `obj` contains a `Cat` object, `toString` returns "Meow! Meow!"

Polymorphism

```
import java.util.Scanner;

public class Pets
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Object obj;

    System.out.print("Which type of pet do you prefer?\n" +
      "Enter d for dogs or c for cats: ");
    if (stdIn.next().equals("d"))
    {
      obj = new Dog();
    }
    else
    {
      obj = new Cat();
    }
    System.out.println(obj.toString());
    System.out.println(obj);
  } // end main
} // end Pets class
```

Declare `obj` as a generic `Object`.

Polymorphic method call.

Polymorphism

```
public class Dog
{
  public String toString()
  {
    return "Woof! Woof!";
  }
} // end Dog class

public class Cat
{
  public String toString()
  {
    return "Meow! Meow!";
  }
} // end Cat class
```

Dynamic Binding

- Polymorphism is a concept. Dynamic binding is a description of how that concept is implemented.
- More specifically, polymorphism is when different types of objects respond differently to the exact same method call. Dynamic binding is what the JVM does in order to match up a polymorphic method call with a particular method. We'll now describe how that "matching up" process works.
- Just before the JVM executes a method call, it looks at the method call's calling object. More specifically, it looks at the <u>type of the object that's been assigned</u> into the calling object's reference variable. If the assigned object is from class X, the JVM binds class X's method to the method call. If the assigned object is from class Y, the JVM binds class Y's method to the method call. After the JVM binds the appropriate method to the method call, the JVM executes the bound method.

Dynamic Binding Compilation Details

- If `Dog` implements a `display` method that prints "I'm a dog", would the following code work?

```
Object obj = new Dog();
obj.display();
```

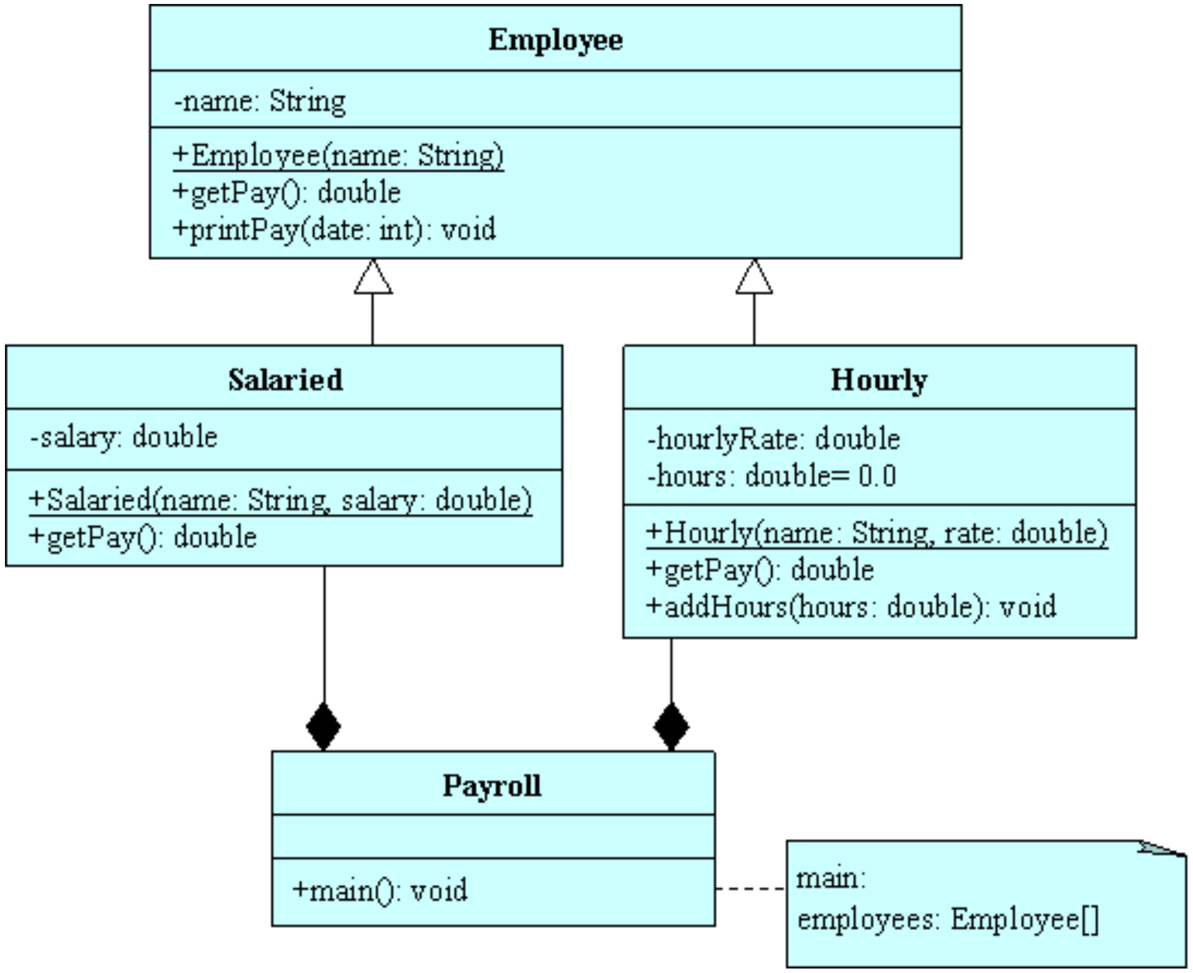
- Be aware of these compiler issues when dynamic binding takes place:
  1. When the compiler sees a method call, *<reference-variable>*`.`*<method-name>*`()`, it checks to see if the reference variable's class contains a method definition for the called method.
  2. Normally, when you assign an object into a reference variable, the object's class and the reference variable's class are the same. But in the above example, note how an object of type `Dog` is assigned into a reference variable of type `Object`. Such assignments only work if the right side's class is a subclass of the left side's class.

Polymorphism with Arrays

- The real usefulness of polymorphism comes when you have an array of generic reference variables and assign different types of objects to different elements in the array.
- That allows you to step through the array and for each array element, you call a polymorphic method.
- At runtime, the JVM uses dynamic binding to pick out the particular methods that apply to the different types of objects that are in the array.
- To illustrate polymorphism with arrays, we present a payroll program that stores payroll data in an employees array….

Polymorphism with Arrays

- UML class diagram for the Payroll program:



Polymorphism with Arrays

```
public class Payroll
{
  public static void main(String[] args)
  {
    Employee[] employees = new Employee[100];
    int day;     // day of week (Sun=0, Mon=1, ..., Sat=6)
    Hourly hourly; // an hourly employee
    employees[0] = new Hourly("Kamal", 25.00);
    employees[1] = new Salaried("Matt", 48000);
    employees[2] = new Hourly("Jason", 20.00);
```

Polymorphism with Arrays

```
// This driver arbitrarily assumes that the payroll's month
// starts on a Tuesday (day = 2) and contains 30 days.

day = 2;
for (int date=1; date<=30; date++)
{
  day++;     // change to the next day of the week
  day %= 7; // causes day of week to cycle from 0-6 repeatedly

  // Loop through all the employees
  for (int i=0; i<employees.length && employees[i] != null; i++)
  {
```

Polymorphism with Arrays

```
if (day > 0 && day < 6 && employees[i] instanceof Hourly)
{
  hourly = (Hourly) employees[i];
  hourly.addHours(8);
}
```

The `instanceof` operator returns true if the object at its left is an instance of the class at its right.

The cast operator is necessary because without it, you'd get a compilation error.

Why? Because we're attempting to assign a superclass-declared object into a subclass reference variable (employees is declared with an `Employee` superclass type and hourly is declared with an `Hourly` subclass type).

If you want to assign a superclass object into a superclass reference variable, you can do it, but only if the "superclass object" really contains a subclass object and you include a cast operator.

Polymorphism with Arrays

```
        // Print hourly employee paychecks on Fridays.
        // Print salaried employee paychecks on 15th and 30th.
        if ((day == 5 && employees[i] instanceof Hourly) ||
          (date%15 == 0 && employees[i] instanceof Salaried))
        {
          employees[i].printPay(date);
        }
      } // end for i
    } // end for date
  } // end main
} // end class Payroll
```

Polymorphism with Arrays

```
public class Employee
{
  private String name;

  //*****************************************************

  public Employee(String name)
  {
    this.name = name;
  }

  //*****************************************************

  public void printPay(int date)
  {
    System.out.printf("%2d %10s: %8.2f\n", date, name, getPay());
  } // end printPay

  //*****************************************************

  // This dummy method is needed to satisfy the compiler.

  public double getPay()
  {
    System.out.println("error! in dummy");
    return 0.0;
  } // end getPay
} // end class Employee
```

polymorphic method call

This method never executes; it's provided to satisfy the compiler.

Polymorphism with Arrays

```
public class Salaried extends Employee
{
  private double salary;

  //************************************************************

  public Salaried(String name, double salary)
  {
    super(name);
    this.salary = salary;
  } // end constructor

  //************************************************************

  public double getPay()
  {
    return this.salary / 24;
  } // end getPay
} // end class Salaried
```

Polymorphism with Arrays

```
public class Hourly extends Employee
{
  private double hourlyRate;
  private double hours = 0.0;

  //***********************************************************

  public Hourly(String name, double rate)
  {
    super(name);
    hourlyRate = rate;
  } // end constructor

  //***********************************************************

  public double getPay()
  {
    double pay = hourlyRate * hours;
    hours = 0.0;
    return pay;
  } // end getPay

  //***********************************************************

  public void addHours(double hours)
  {
    this.hours += hours;
  } // end addHours
} // end class Hourly
```

`abstract` Methods and Classes

- Declare a method to be `abstract` if the method's class is a superclass and the method is merely a "dummy" method for an overriding method(s) in a subclass(es).
- Java requires that when you define a method to be `abstract`, you must:
  - Use an *abstract method heading* instead of a method definition. An abstract method heading is the same as a standard method heading except that it includes the `abstract` modifier and a trailing semicolon.
  - Define an overriding version of that method in each of the superclass's subclasses.
  - Define the superclass to be `abstract` by using the `abstract` modifier.
- In defining a class to be `abstract`, you're telling the compiler to not allow the class to be instantiated; i.e., if a program attempts to instantiate an `abstract` class, a compilation error will be generated.

`abstract` Methods and Classes

```
public abstract class Employee
{
  private String name;

  public abstract double getPay();

  //*****************************************************

  public Employee(String name)
  {
    this.name = name;
  }

  //*****************************************************

  public void printPay(int date)
  {
    System.out.printf("%2d %10s: %8.2f\n", date, name, getPay());
  } // end printPay
} // end class Employee
```