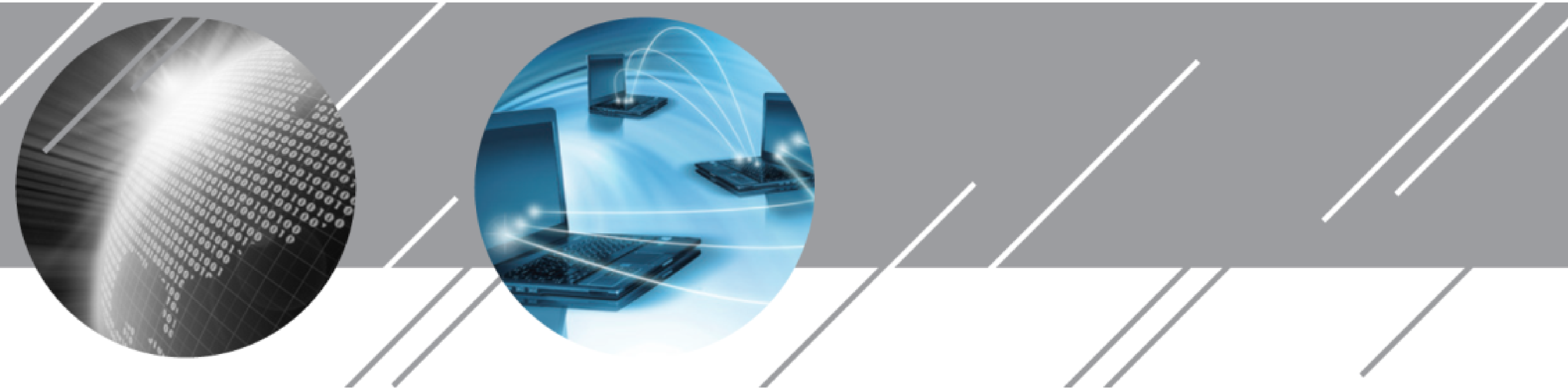
Object Oriented Programming

Introduction to Java

Ch. 6. More About Objects and Methods

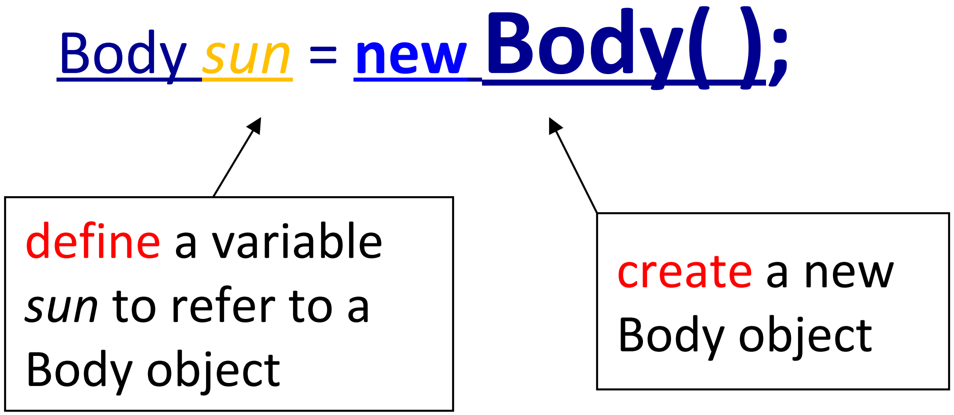


Dept. of Software, Gachon University
Ahyoung Choi, Spring

6.1 Constructors

Object Creation

```
class Body {
  private long idNum;
  private String name;
  private Body orbits;
  private static long nextID = 0;
}
```



Body *sun* = **new** **Body( );**

define a variable *sun* to refer to a Body object

create a new Body object

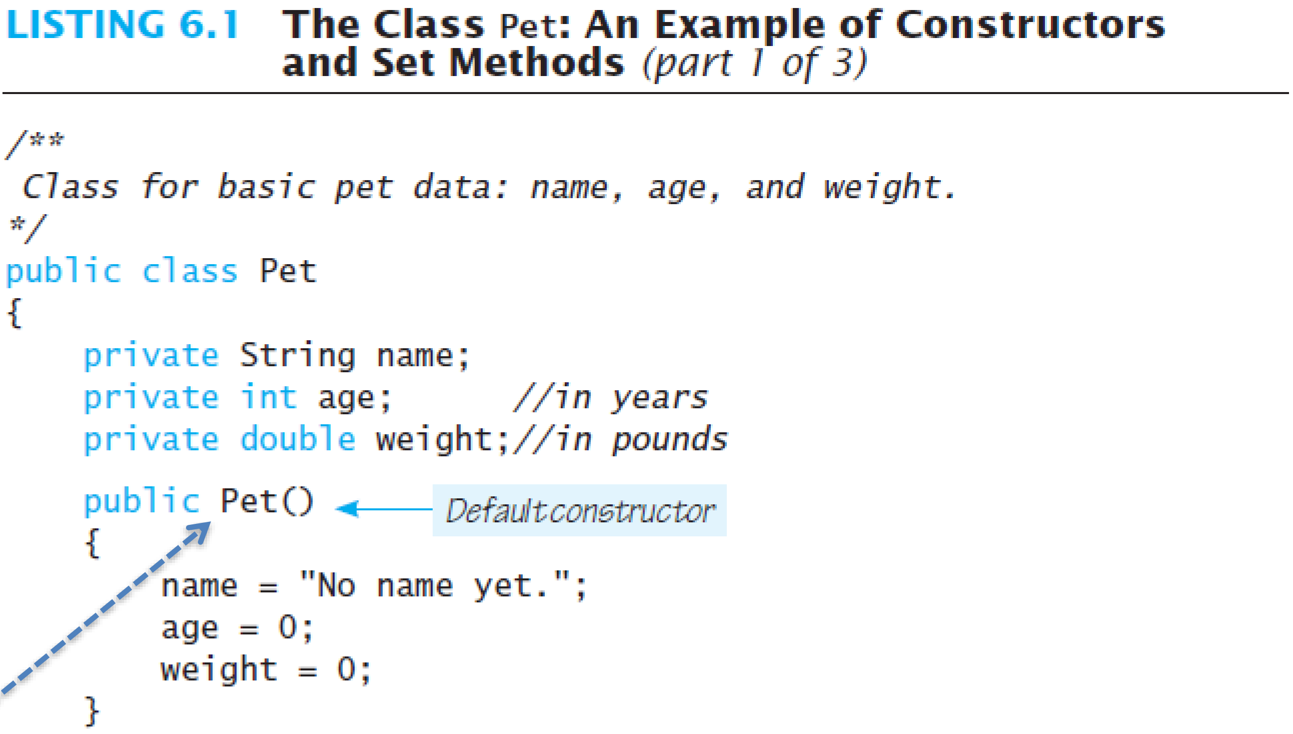
Question:

Can we initialize the member variables of the new object *sun* before its first use?

Constructors

- **constructor**
  - **a way to initialize** an object before the reference to the object is returned by **`new`**
  - has the **same name as the class**
  - can **have parameters**
    - To specify initial values if desired
  - may have **multiple definitions**
    - Each with different numbers or types of parameters

Default constructor



LISTING 6.1 The Class Pet: An Example of Constructors and Set Methods *(part 1 of 3)*

```
/**
 Class for basic pet data: name, age, and weight.
*/
public class Pet
{
    private String name;
    private int age;      //in years
    private double weight;//in pounds

    public Pet()
    {
        name = "No name yet.";
        age = 0;
        weight = 0;
    }
```

Default constructor

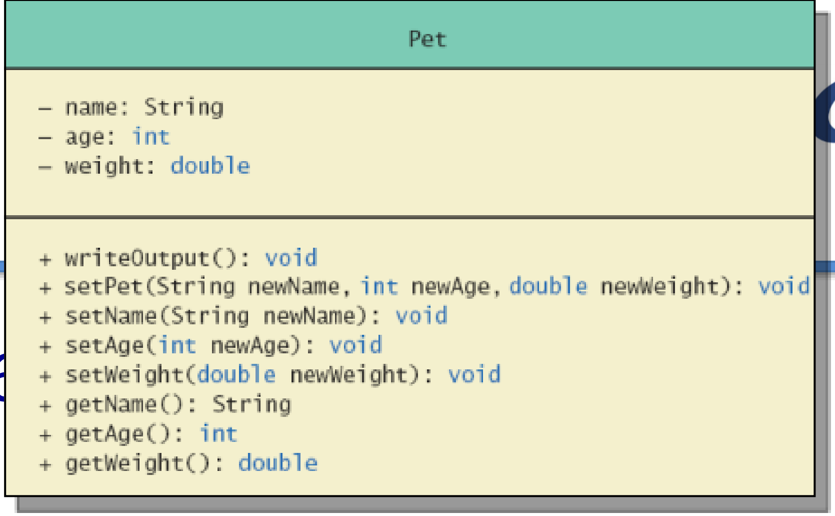
- No return type (e.g., void, int) is required!!

Default constructor

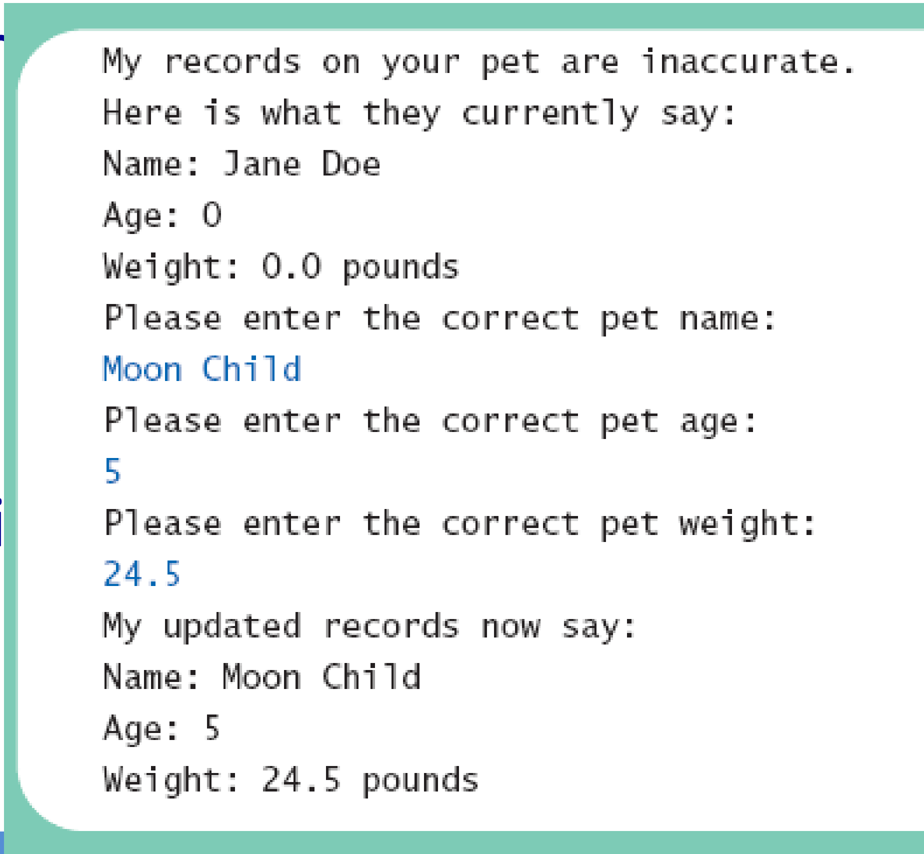
- Default constructor
  - **Constructor without parameters**
  - Java will define this automatically if the class designer does not define any constructors
  - If you do define a constructor, Java will not automatically define a default constructor
- Usually default constructors not included in class diagram

Lab: pet class

- Note sample code, listing 6
  class Pet
- Note different constructor
  - Default
  - With 3 parameters
  - With String parameter
  - With double parameter
- Note sample program, listi
  class PetDemo



```
Pet

- name: String
- age: int
- weight: double

+ writeOutput(): void
+ setPet(String newName, int newAge, double newWeight): void
+ setName(String newName): void
+ setAge(int newAge): void
+ setWeight(double newWeight): void
+ getName(): String
+ getAge(): int
+ getWeight(): double
```



```
My records on your pet are inaccurate.
Here is what they currently say:
Name: Jane Doe
Age: 0
Weight: 0.0 pounds
Please enter the correct pet name:
Moon Child
Please enter the correct pet age:
5
Please enter the correct pet weight:
24.5
My updated records now say:
Name: Moon Child
Age: 5
Weight: 24.5 pounds
```

```
public Pet(String initialName, int initialAge,
           double initialWeight)
{
}
public void setPet(String newName, int newAge,
                   double newWeight)
{
    name = newName;
    if ((newAge < 0) || (newWeight < 0))
    {
        System.out.println("Error: Negative age or weight.");
        System.exit(0);
    }
    else
    {
        age = newAge;
        weight = newWeight;
    }
}
```

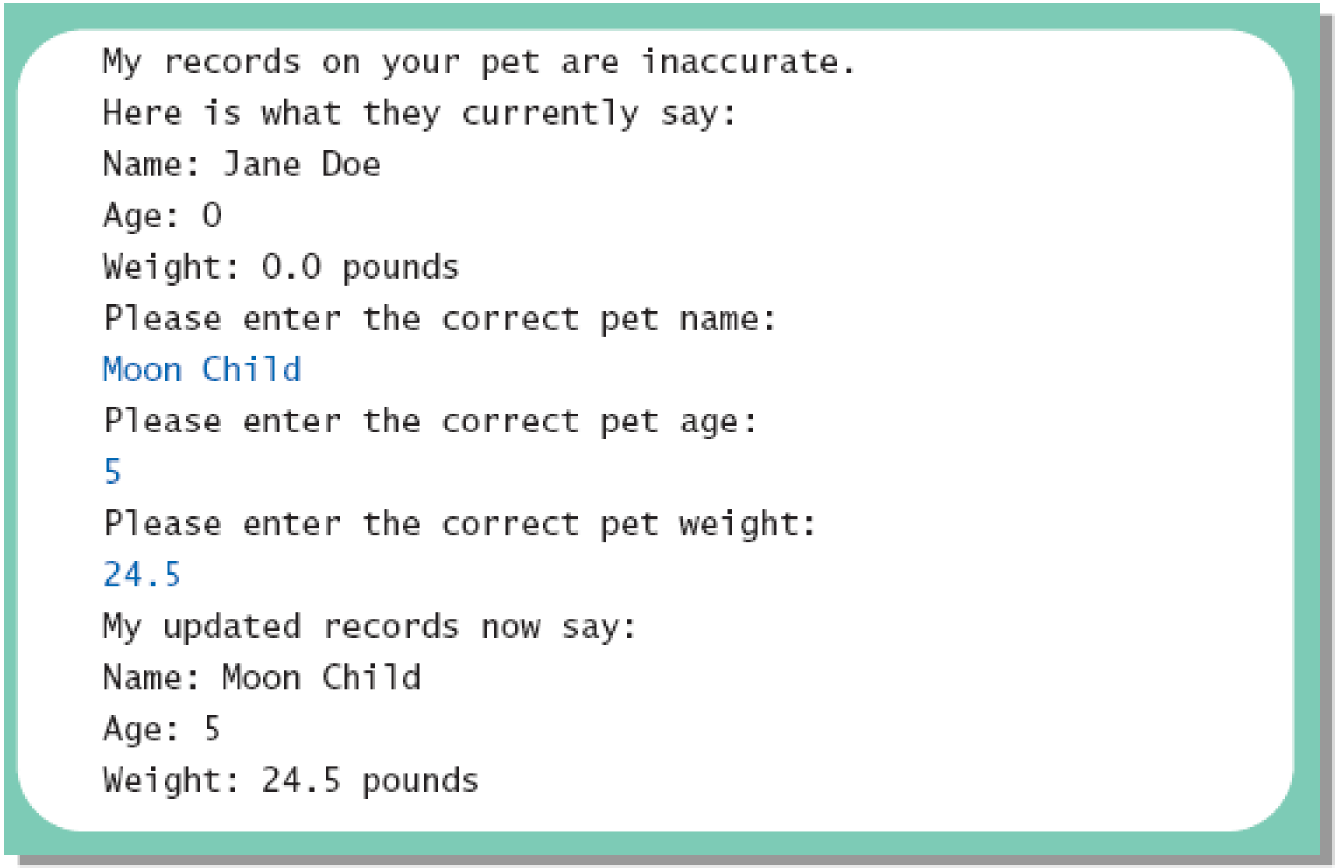
```
public Pet(String initialName)
{
}

public void setName(String newName)
{
    name = newName; //age and weight are unchanged.
}
```

```
public Pet(int initialAge)
{
}

public void setAge(int newAge)
{
    if (newAge < 0)
    {
        System.out.println("Error: Negative age.");
        System.exit(0);
    }
    else
        age = newAge;
    //name and weight are unchanged.
}
```

```
public Pet(double initialWeight)
{
[redacted]
}
public void setWeight(double newWeight)
{
    if (newWeight < 0)
    {
        System.out.println("Error: Negative weight.");
        System.exit(0);
    }
    else
        weight = newWeight; //name and age are unchanged.
}
```

```
    public String getName()
    {
        return name;
    }
    public int getAge()
    {
        return age;
    }
    public double getWeight()
    {
        return weight;
    }
    public void writeOutput()
    {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age + " years");
        System.out.println("Weight: " + weight + " pounds");
    }
}
```

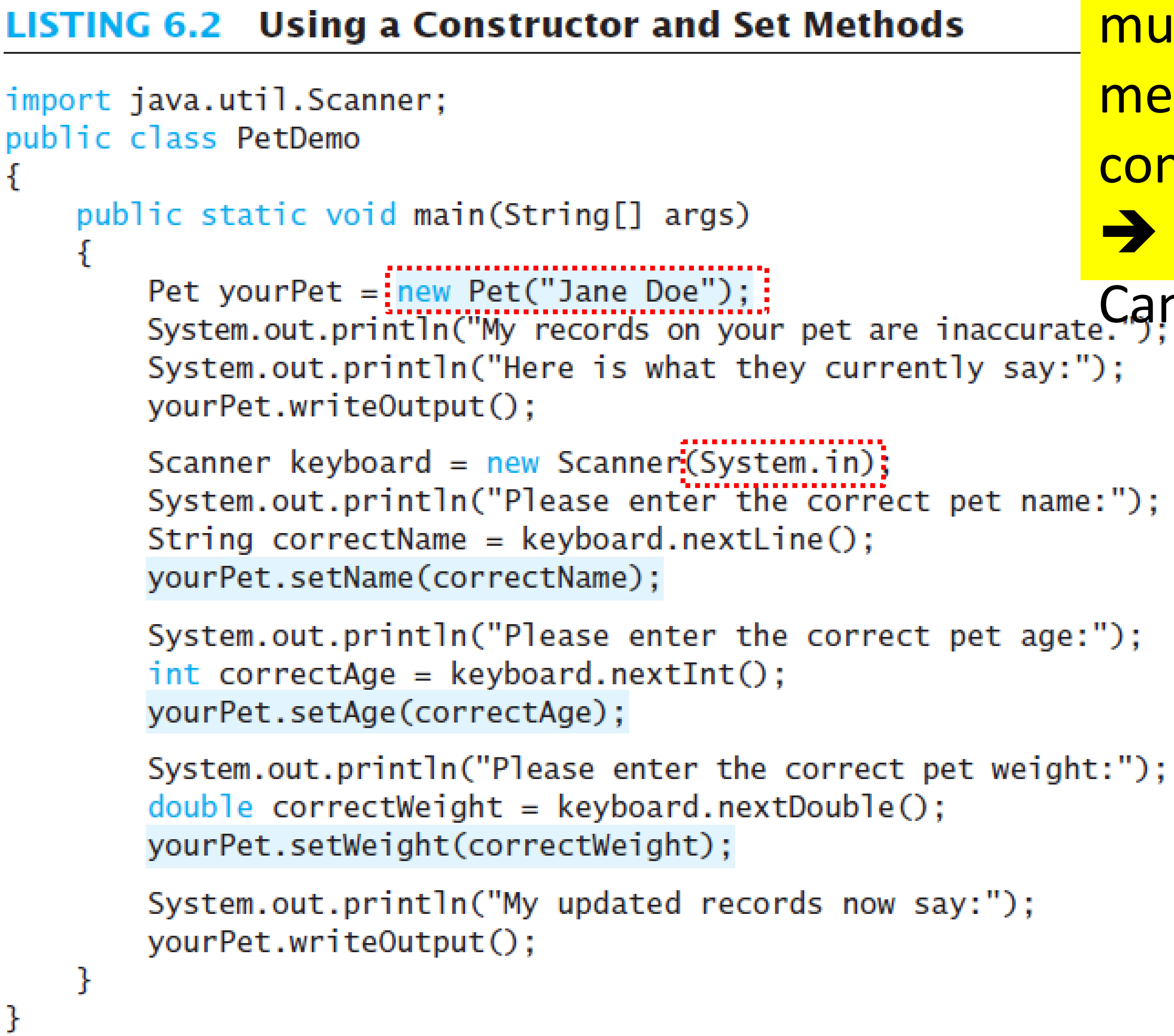
Defining Constructors



```
My records on your pet are inaccurate.
Here is what they currently say:
Name: Jane Doe
Age: 0
Weight: 0.0 pounds
Please enter the correct pet name:
Moon Child
Please enter the correct pet age:
5
Please enter the correct pet weight:
24.5
My updated records now say:
Name: Moon Child
Age: 5
Weight: 24.5 pounds
```

Differences between mutator (setter method) vs constructor?

➔ Can call once vs Can call whenever want

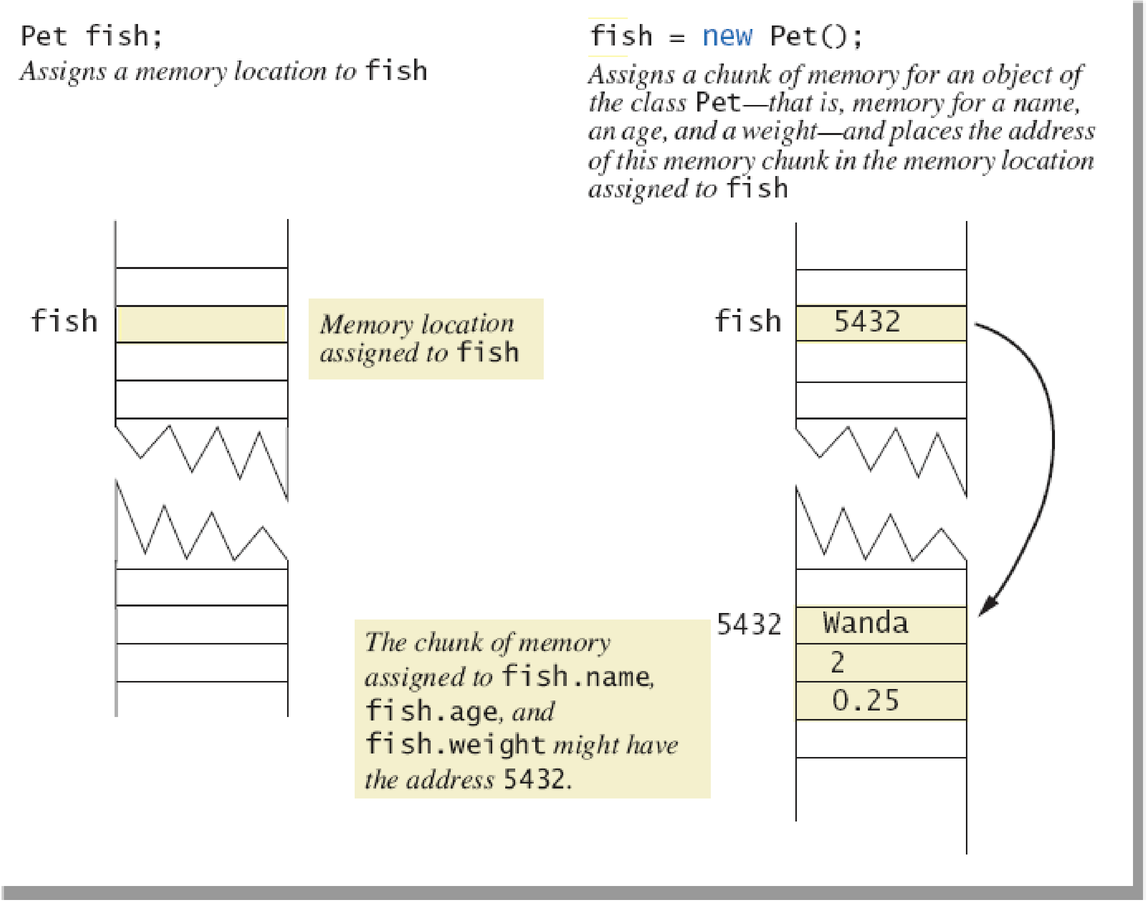


LISTING 6.2 Using a Constructor and Set Methods

```
import java.util.Scanner;
public class PetDemo
{
    public static void main(String[] args)
    {
        Pet yourPet = new Pet("Jane Doe");
        System.out.println("My records on your pet are inaccurate.");
        System.out.println("Here is what they currently say:");
        yourPet.writeOutput();

        Scanner keyboard = new Scanner(System.in);
        System.out.println("Please enter the correct pet name:");
        String correctName = keyboard.nextLine();
        yourPet.setName(correctName);

        System.out.println("Please enter the correct pet age:");
        int correctAge = keyboard.nextInt();
        yourPet.setAge(correctAge);

        System.out.println("Please enter the correct pet weight:");
        double correctWeight = keyboard.nextDouble();
        yourPet.setWeight(correctWeight);

        System.out.println("My updated records now say:");
        yourPet.writeOutput();
    }
}
```

Defining Constructors

- Figure 6.2 A constructor returning a reference



`Pet fish;`

Assigns a memory location to `fish`

`fish = new Pet();`

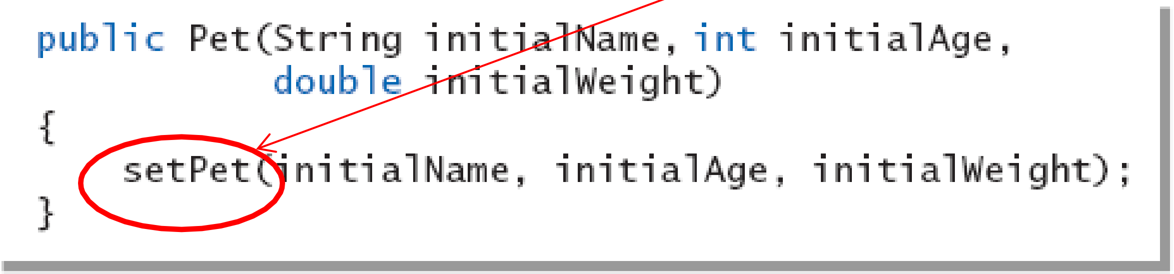
Assigns a chunk of memory for an object of the class `Pet`*—that is, memory for a name, an age, and a weight—and places the address of this memory chunk in the memory location assigned to* `fish`

Memory location assigned to `fish`

The chunk of memory assigned to `fish.name`*,* `fish.age`*, and* `fish.weight` *might have the address* `5432`*.*

Calling Methods from Other Constructors

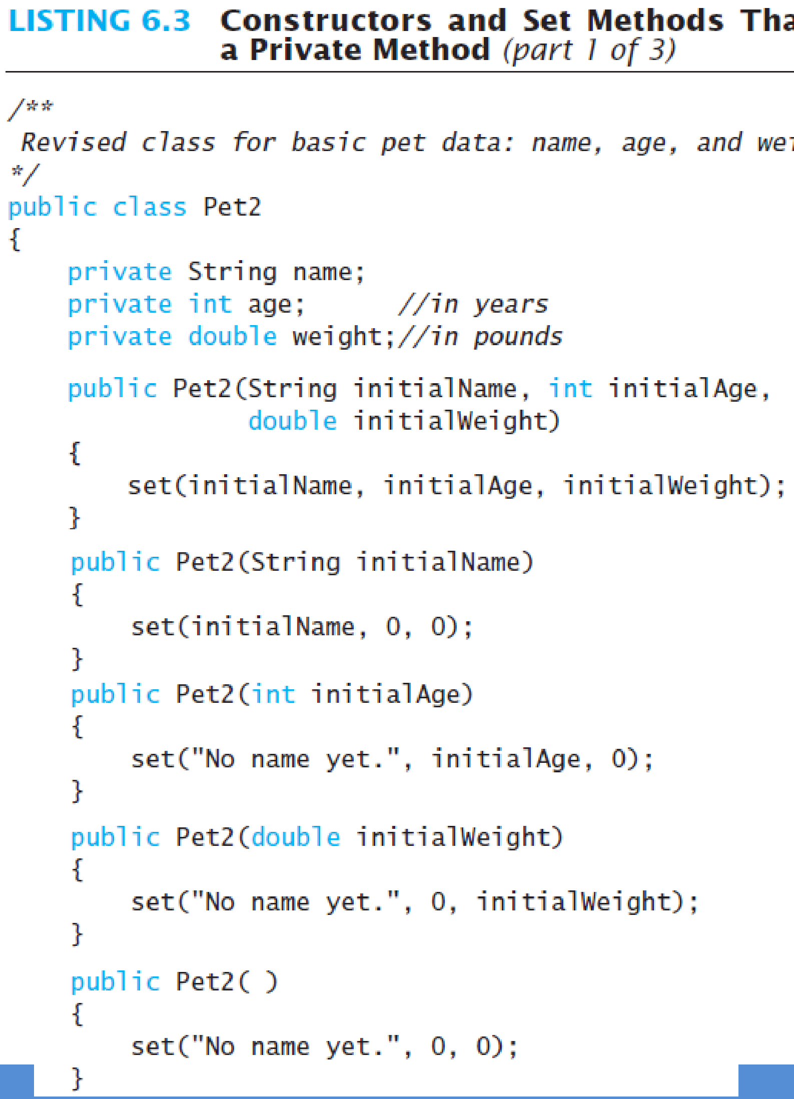
- Method can call other methods within a classes.
- **Similarly constructor can call methods within a class**



```
public Pet(String initialName, int initialAge,
           double initialWeight)
{
    setPet(initialName, initialAge, initialWeight);
}
```

- Keeps from (avoid) repeating code

Lab: pet class – Let’s make it simple



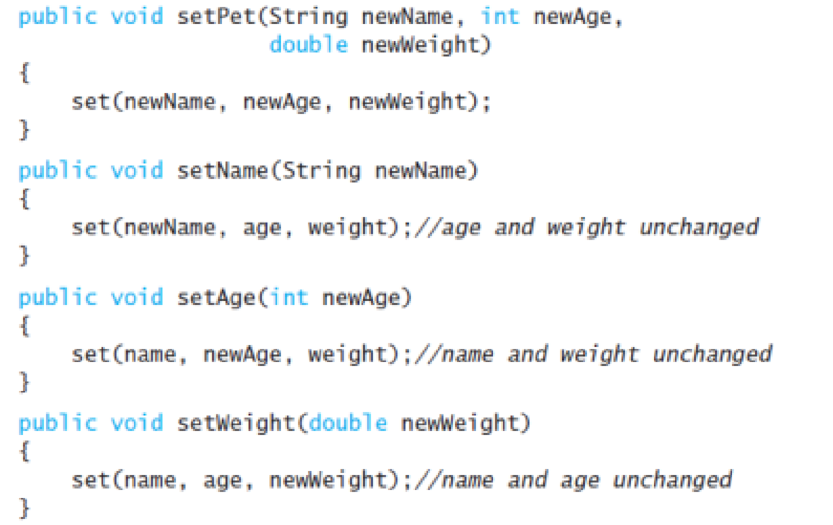
LISTING 6.3 **Constructors and Set Methods That Call a Private Method** *(part 1 of 3)*

```
/**
 Revised class for basic pet data: name, age, and weight.
*/
public class Pet2
{
    private String name;
    private int age;      //in years
    private double weight;//in pounds

    public Pet2(String initialName, int initialAge,
                double initialWeight)
    {
        set(initialName, initialAge, initialWeight);
    }
    public Pet2(String initialName)
    {
        set(initialName, 0, 0);
    }
    public Pet2(int initialAge)
    {
        set("No name yet.", initialAge, 0);
    }
    public Pet2(double initialWeight)
    {
        set("No name yet.", 0, initialWeight);
    }
    public Pet2( )
    {
        set("No name yet.", 0, 0);
    }
```

View sample code, listing 6.3

class Pet2

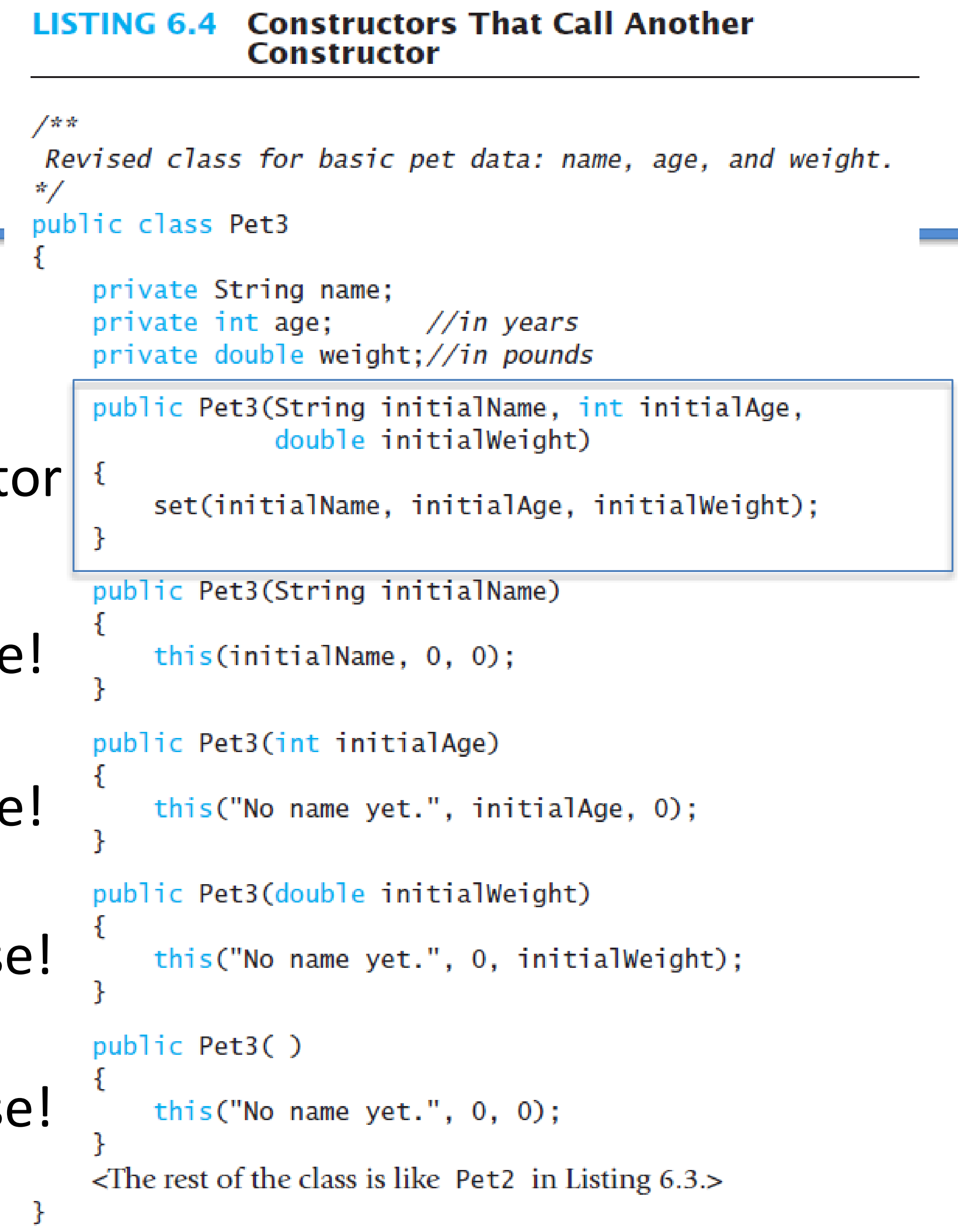
```
private void set(String newName, int newAge, double newWeight){
      name = newName;
      if ((newAge < 0) || (newWeight < 0)){
            System.out.println("Error: Negative age or weight.");
            System.exit(0);
      }
      else{
            age = newAge;
            weight = newWeight;
      }
}
```



```
    public void setPet(String newName, int newAge,
                       double newWeight)
    {
        set(newName, newAge, newWeight);
    }
    public void setName(String newName)
    {
        set(newName, age, weight);//age and weight unchanged
    }
    public void setAge(int newAge)
    {
        set(name, newAge, weight);//name and weight unchanged
    }
    public void setWeight(double newWeight)
    {
        set(name, age, newWeight);//name and age unchanged
    }
```

Calling Constructor from Other Constructors

- Use initial constructor and method set
- In the other constructors use the this reference to call initial constructor
- View revised class, listing 6.4 class Pet3
  - Note calls to initial constructor



LISTING 6.4 Constructors That Call Another Constructor

```
/**
 Revised class for basic pet data: name, age, and weight.
*/
public class Pet3
{
    private String name;
    private int age;      //in years
    private double weight;//in pounds
```

Initial constructor

```
    public Pet3(String initialName, int initialAge,
                double initialWeight)
    {
        set(initialName, initialAge, initialWeight);
    }
```

Use!

```
    public Pet3(String initialName)
    {
        this(initialName, 0, 0);
    }
```

Use!

```
    public Pet3(int initialAge)
    {
        this("No name yet.", initialAge, 0);
    }
```

Use!

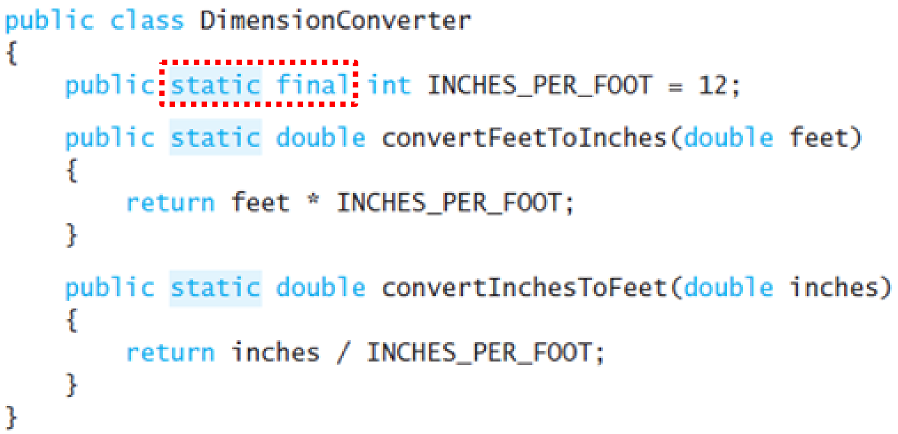
```
    public Pet3(double initialWeight)
    {
        this("No name yet.", 0, initialWeight);
    }
```

Use!

```
    public Pet3( )
    {
        this("No name yet.", 0, 0);
    }
    <The rest of the class is like Pet2 in Listing 6.3.>
}
```

6.2 Static Variables and Static Methods

Question



```
public class DimensionConverter
{
    public static final int INCHES_PER_FOOT = 12;

    public static double convertFeetToInches(double feet)
    {
        return feet * INCHES_PER_FOOT;
    }

    public static double convertInchesToFeet(double inches)
    {
        return inches / INCHES_PER_FOOT;
    }
}
```

- What does “`static final`” mean ?
  - What is it implemented ?
  - If you generate N objects of a class DimemsionCoverter, how many variable INCHES_PER_FOOT are generated ?

Recall

- Recall that “classes do not have data; individual objects have data”

- This is not always true – classes can have data, too
  - **static** variables and methods **belong to a class as a whole**, not to an individual object
  - When would you want a method that does not need an object?
    - If the method perform a general function instead of actions on an object

Static Variables

- **Static means "belonging to *the class* in general", not to an *individual object***

- A variable may be declared with the **static** keyword
  - `e.g. static int numTicketsSold;`
  - There *is one* variable numTickets for the class *not one per object!!!*

- **Static variables are shared by all objects of a class**
  - Variables declared `static final` are considered constants – value cannot be changed

Static Variables

- Variables declared **`static`** (without **`final`**) can be changed
  - Only one instance of the variable exists
  - It can be accessed (**shared**) by all instances of the class

- A public static variable may be accessed by
  - *ClassName.variableName*
  - E.g. `Math.PI`

Static Methods

- **Some methods** may have no relation to any type of object
- Example
  - Compute max of two integers
  - Convert character from upper- to lower case

- **A method may be declared with the *static* keyword**
- **Static method declared in a class**
  - **Can be invoked without using an object**
  - Static methods live at *class level*, not at *object level*

Static Methods (contd..)

- A static method that is public can be accessed
  - ClassName.methodName(args)

```
double result = Math.sqrt(25.0);
int numSold = Ticket.getNumberSold();
```

- Static methods access static variables and methods, but not instance variables

```
public static int getNumSold(){
        return numTicketsSold;
}
```

Problem?

```
public class JustAdd {
    int x;
    int y;
    int z;

    public static void main(String args[]) {
        x = 5;
        y = 10;
        z = x + y;
    }
}
```

all are wrong

Static method - System class

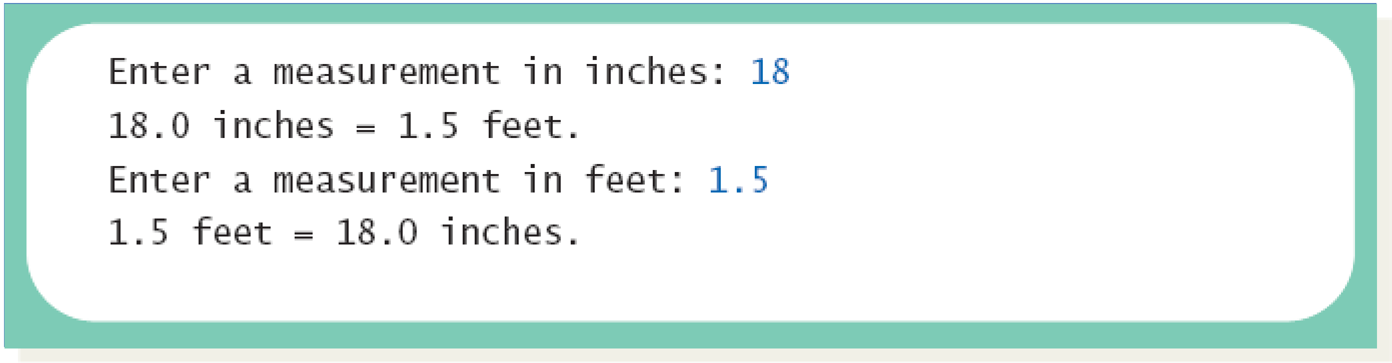
- Facilities provided by System
  - Standard output
  - Error output streams
  - Standard input and access to externally defined properties and environment variables.
  - A means of loading files and libraries
- It cannot be instantiated → defined with "static"

Field definition	Explaination
static **PrintStream err**	This is the "standard" error output stream.
static **InputStream in**	This is the "standard" input stream.
static **PrintStream out**	This is the "standard" output stream.

A static variable is common to all the instances (or objects) of the class because it is a class level variable. Only a single copy of static variable is created and shared among all the instances of the class.

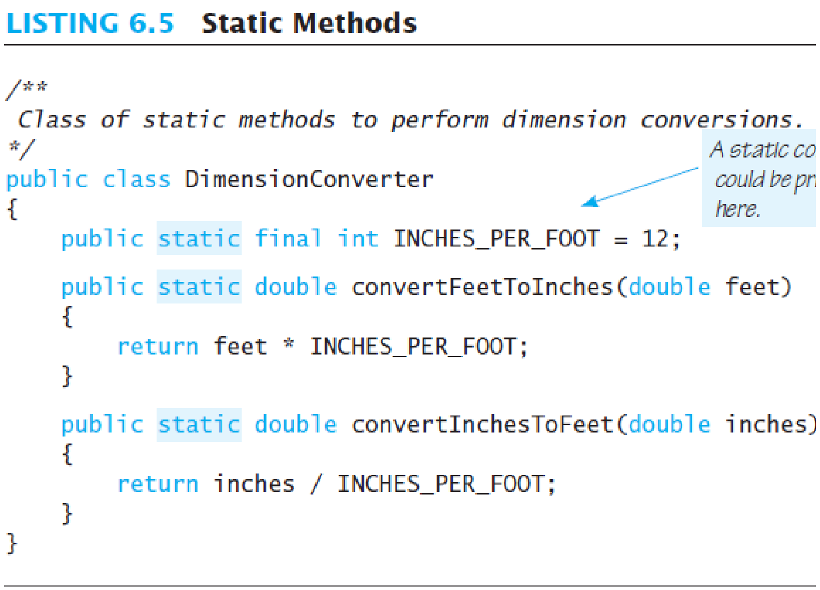
Lab: Static Methods

- View sample class, listing 6.5
  `class DimensionConverter`
- View demonstration program, listing 6.6
  `class DimensionConverterDemo`



```
Enter a measurement in inches: 18
18.0 inches = 1.5 feet.
Enter a measurement in feet: 1.5
1.5 feet = 18.0 inches.
```

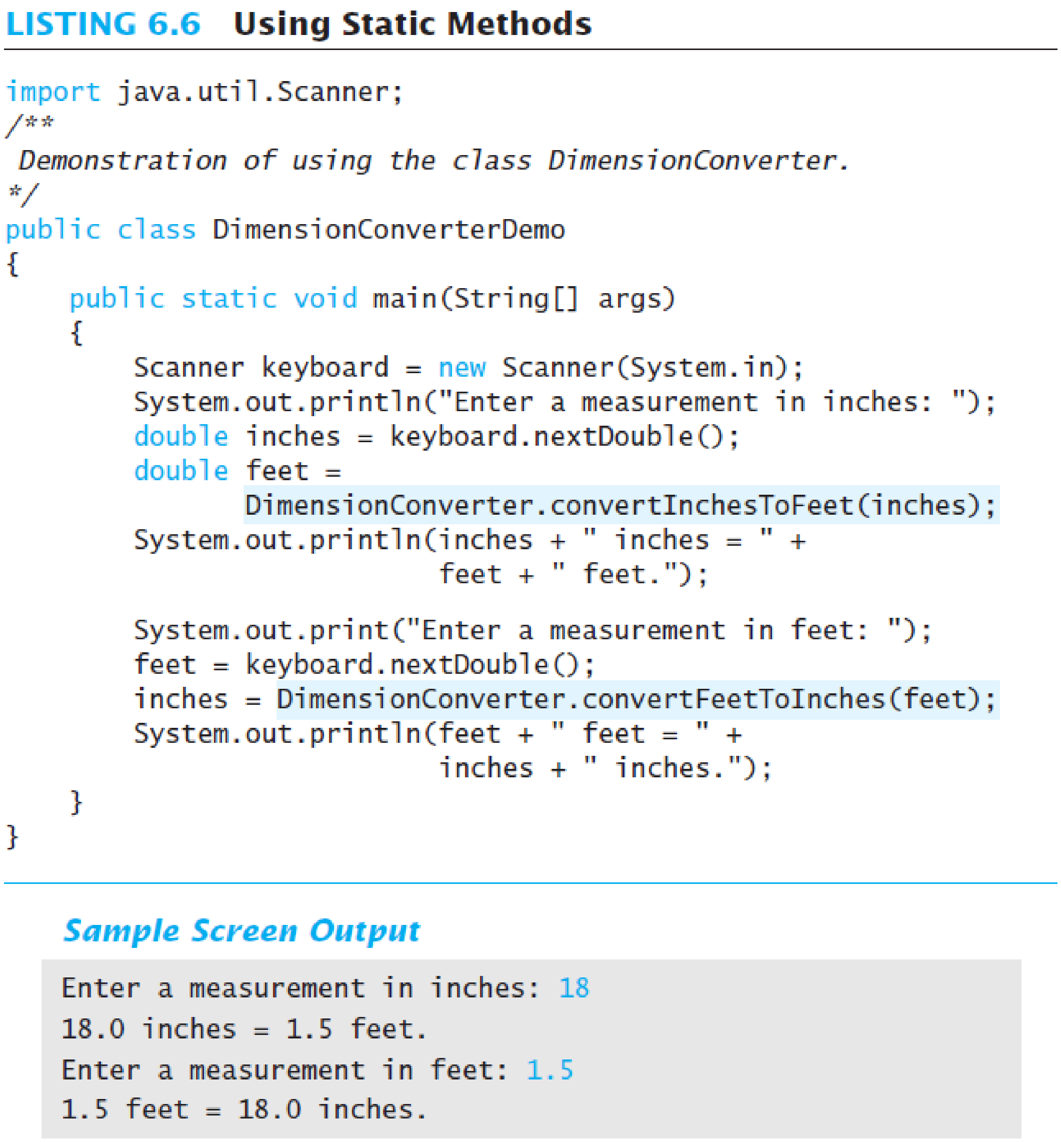
6.5-6.6



LISTING 6.5 Static Methods

```
/**
 Class of static methods to perform dimension conversions.
*/
public class DimensionConverter
{
    public static final int INCHES_PER_FOOT = 12;

    public static double convertFeetToInches(double feet)
    {
        return feet * INCHES_PER_FOOT;
    }

    public static double convertInchesToFeet(double inches)
    {
        return inches / INCHES_PER_FOOT;
    }
}
```

A static co
could be pr
here.



LISTING 6.6 Using Static Methods

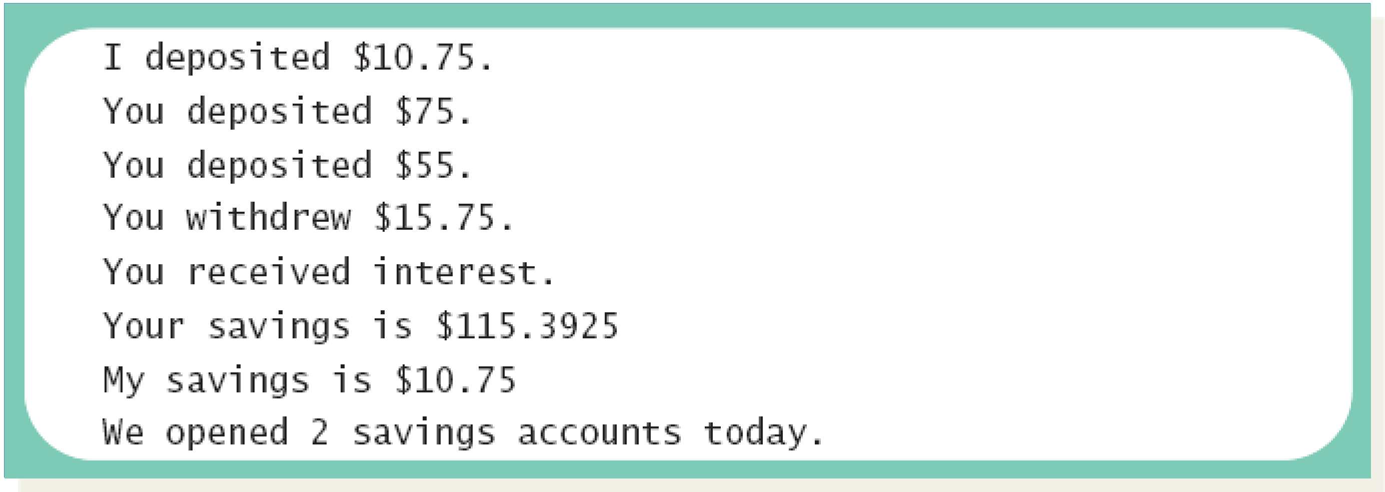
```
import java.util.Scanner;
/**
 Demonstration of using the class DimensionConverter.
*/
public class DimensionConverterDemo
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter a measurement in inches: ");
        double inches = keyboard.nextDouble();
        double feet =
                DimensionConverter.convertInchesToFeet(inches);
        System.out.println(inches + " inches = " +
                           feet + " feet.");

        System.out.print("Enter a measurement in feet: ");
        feet = keyboard.nextDouble();
        inches = DimensionConverter.convertFeetToInches(feet);
        System.out.println(feet + " feet = " +
                           inches + " inches.");
    }
}
```

Sample Screen Output

```
Enter a measurement in inches: 18
18.0 inches = 1.5 feet.
Enter a measurement in feet: 1.5
1.5 feet = 18.0 inches.
```

Lab: Mixing Static and Nonstatic Methods

- View sample class, listing 6.7
  `class SavingsAccount`
- View demo program, listing 6.8
  `class SavingsAccountDemo`



```
I deposited $10.75.
You deposited $75.
You deposited $55.
You withdrew $15.75.
You received interest.
Your savings is $115.3925
My savings is $10.75
We opened 2 savings accounts today.
```

```
import java.util.Scanner;
public class SavingsAccount {
    private double balance;
    public static double interestRate = 0;
    public static int numberOfAccounts = 0;
    public SavingsAccount ()    {
        balance = 0;
        numberOfAccounts++;
    }
    public static void setInterestRate (double newRate)    {
        interestRate = newRate;
    }
    public static double getInterestRate ()    {
        return interestRate;
    }
    public static double getNumberOfAccounts ()    {
        return numberOfAccounts;
    }
    public void deposit (double amount)    {

    }
    public double withdraw (double amount)    {



    }
    public void addInterest ()    {

    }
    public double getBalance ()    {   return balance;    }
    public static void showBalance (SavingsAccount account)    {
        System.out.print (account.getBalance ());
    }
}
```

외부 공유 필요한 변수를 Static 변수 선언

생성자 선언 및 초기화

Static 변수 접근을 위한 함수는 static method 선언
➔ Interest rate 값 의 setter getter 함수 구현
➔ 전체 account number 변수를 위한 setter, getter 구현

함수 : 입금 함수

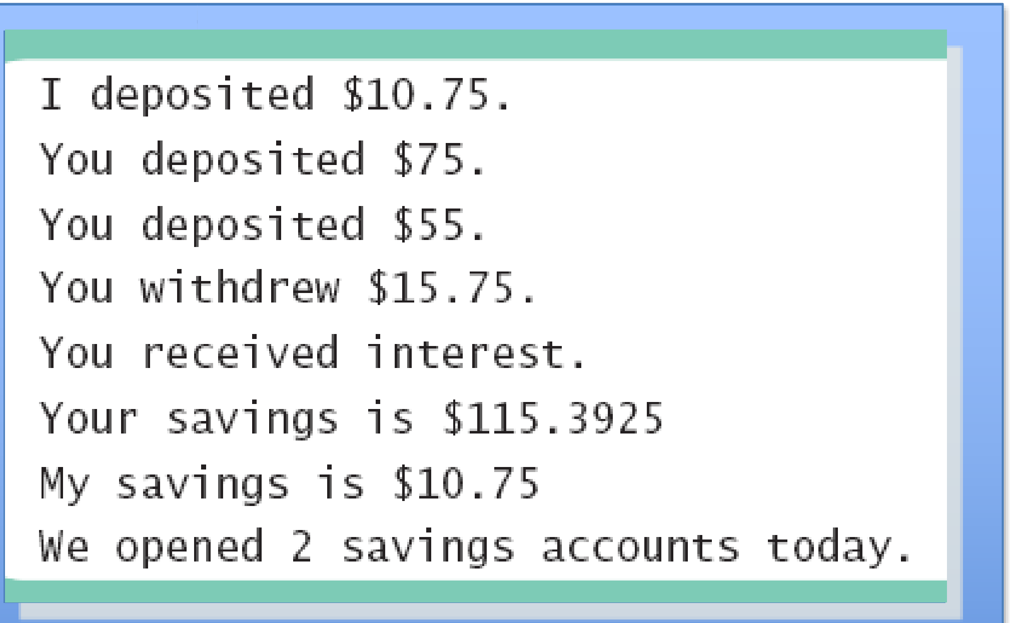
함수 : 출금 계산 (금액 출금 후 잔액 저장)

함수 : 이율계산

함수 : balance 변수 getter 함수

함수 : balance 출력

```
public class SavingsAccountDemo {
   public static void main (String [] args)    {
```

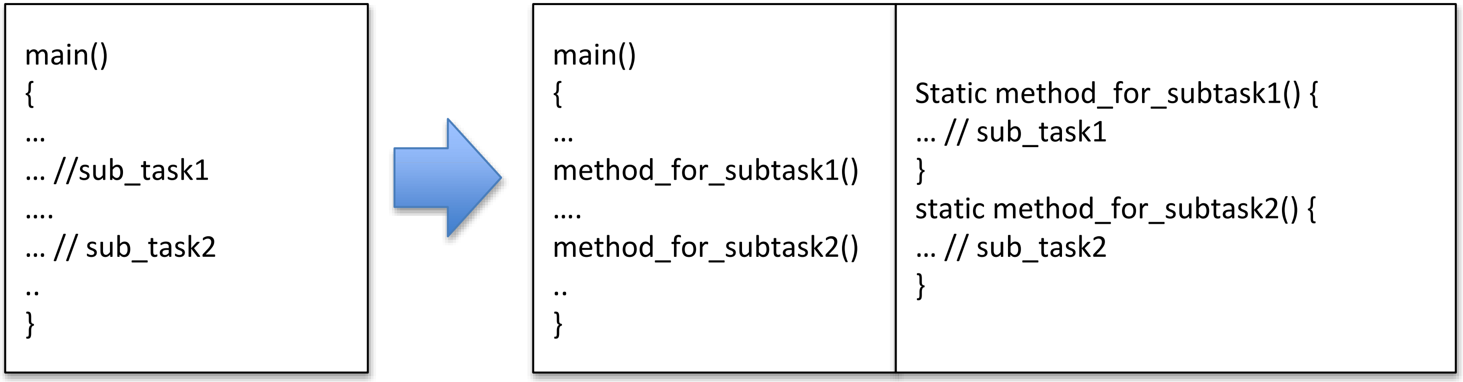


```
I deposited $10.75.
You deposited $75.
You deposited $55.
You withdrew $15.75.
You received interest.
Your savings is $115.3925
My savings is $10.75
We opened 2 savings accounts today.
```

```
                                                  // 이율 set
                                                  // account 인스턴스 2개 생성

                                                  // 첫번째 계좌에 10.75불 입금

                                                  // 두번째 계좌에 75불 입금

                                                 // 두번째 계좌에서 15불 출금

                                                // 예금액 100불 이상이면 이율계산


                                            alance ());  // 두번째 계좌 발란스 확인

                                                     // 첫번째 계좌 발란스 확인


                                                     // 총 개설 account 확인
                                            nts today.");

}
```

Tasks of `main` in Subtasks

- Program may have
  - Complicated logic
  - Repetitive code
- Create static methods to accomplish subtasks



```
main()
{
…
… //sub_task1
….
… // sub_task2
..
}
```

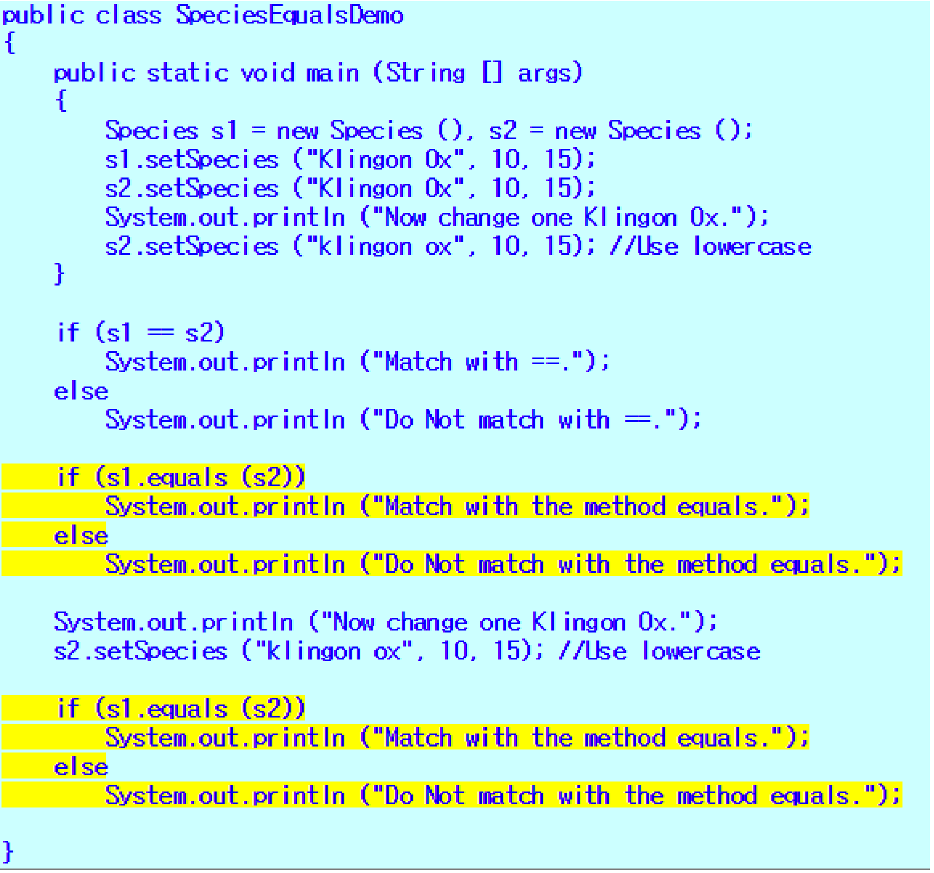
```
main()
{
…
method_for_subtask1()
….
method_for_subtask2()
..
}
```

```
Static method_for_subtask1() {
… // sub_task1
}
static method_for_subtask2() {
… // sub_task2
}
```

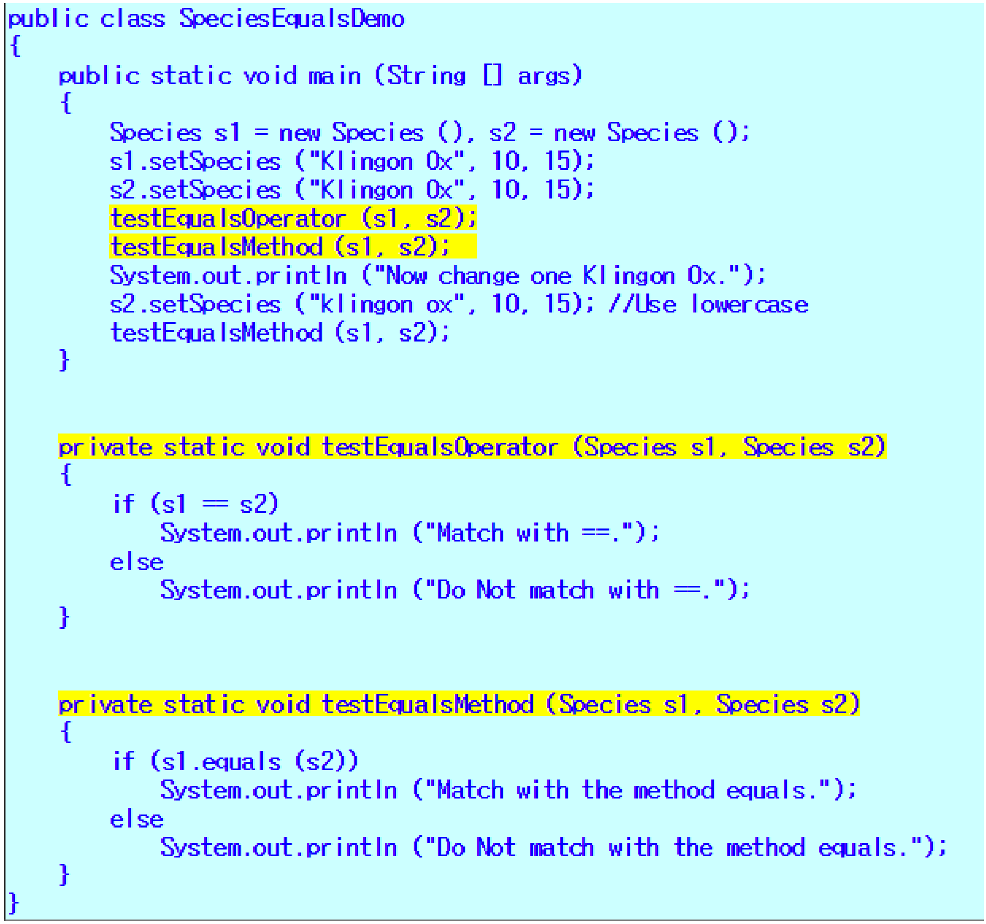
- Consider example code, listing 6.9
  a `main` method with repetitive code
- Note alternative code, listing 6.10
  uses helping methods

Adding Method `main` to a Class

- Method main used so far in its own class within a separate file
- Often useful to include method main within class definition
  - To create objects in other classes
  - To be run as a program
- Note example code, listing 6.11 a redefined `class Species`
  - When used as ordinary class, method `main` ignored



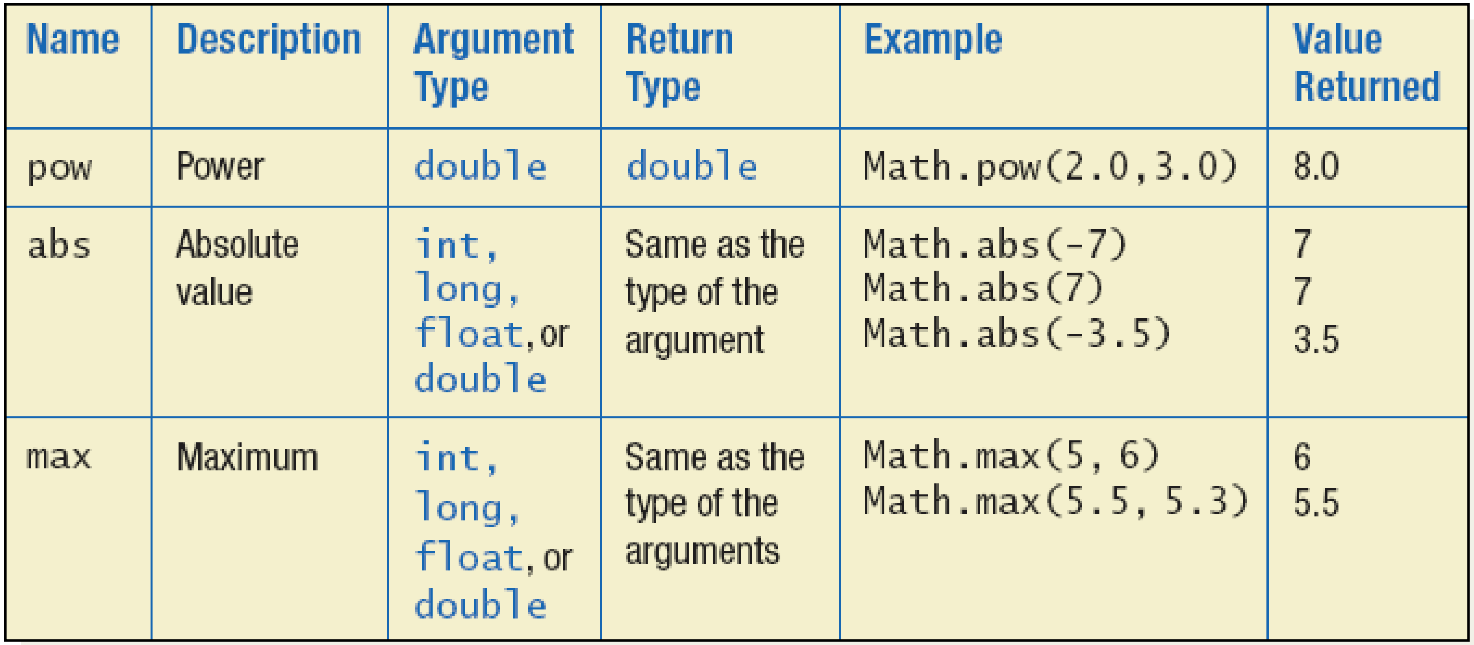
```
public class SpeciesEqualsDemo
{
    public static void main (String [] args)
    {
        Species s1 = new Species (), s2 = new Species ();
        s1.setSpecies ("Klingon Ox", 10, 15);
        s2.setSpecies ("Klingon Ox", 10, 15);
        System.out.println ("Now change one Klingon Ox.");
        s2.setSpecies ("klingon ox", 10, 15); //Use lowercase
    }

    if (s1 == s2)
        System.out.println ("Match with ==.");
    else
        System.out.println ("Do Not match with ==.");

    if (s1.equals (s2))
        System.out.println ("Match with the method equals.");
    else
        System.out.println ("Do Not match with the method equals.");

    System.out.println ("Now change one Klingon Ox.");
    s2.setSpecies ("klingon ox", 10, 15); //Use lowercase

    if (s1.equals (s2))
        System.out.println ("Match with the method equals.");
    else
        System.out.println ("Do Not match with the method equals.");

}
```



```
public class SpeciesEqualsDemo
{
    public static void main (String [] args)
    {
        Species s1 = new Species (), s2 = new Species ();
        s1.setSpecies ("Klingon Ox", 10, 15);
        s2.setSpecies ("Klingon Ox", 10, 15);
        testEqualsOperator (s1, s2);
        testEqualsMethod (s1, s2);
        System.out.println ("Now change one Klingon Ox.");
        s2.setSpecies ("klingon ox", 10, 15); //Use lowercase
        testEqualsMethod (s1, s2);
    }


    private static void testEqualsOperator (Species s1, Species s2)
    {
        if (s1 == s2)
            System.out.println ("Match with ==.");
        else
            System.out.println ("Do Not match with ==.");
    }


    private static void testEqualsMethod (Species s1, Species s2)
    {
        if (s1.equals (s2))
            System.out.println ("Match with the method equals.");
        else
            System.out.println ("Do Not match with the method equals.");
    }
}
```

The Math Class

- Provides many standard mathematical methods
  - Automatically provided, no import needed
- Example methods, figure 6.3a



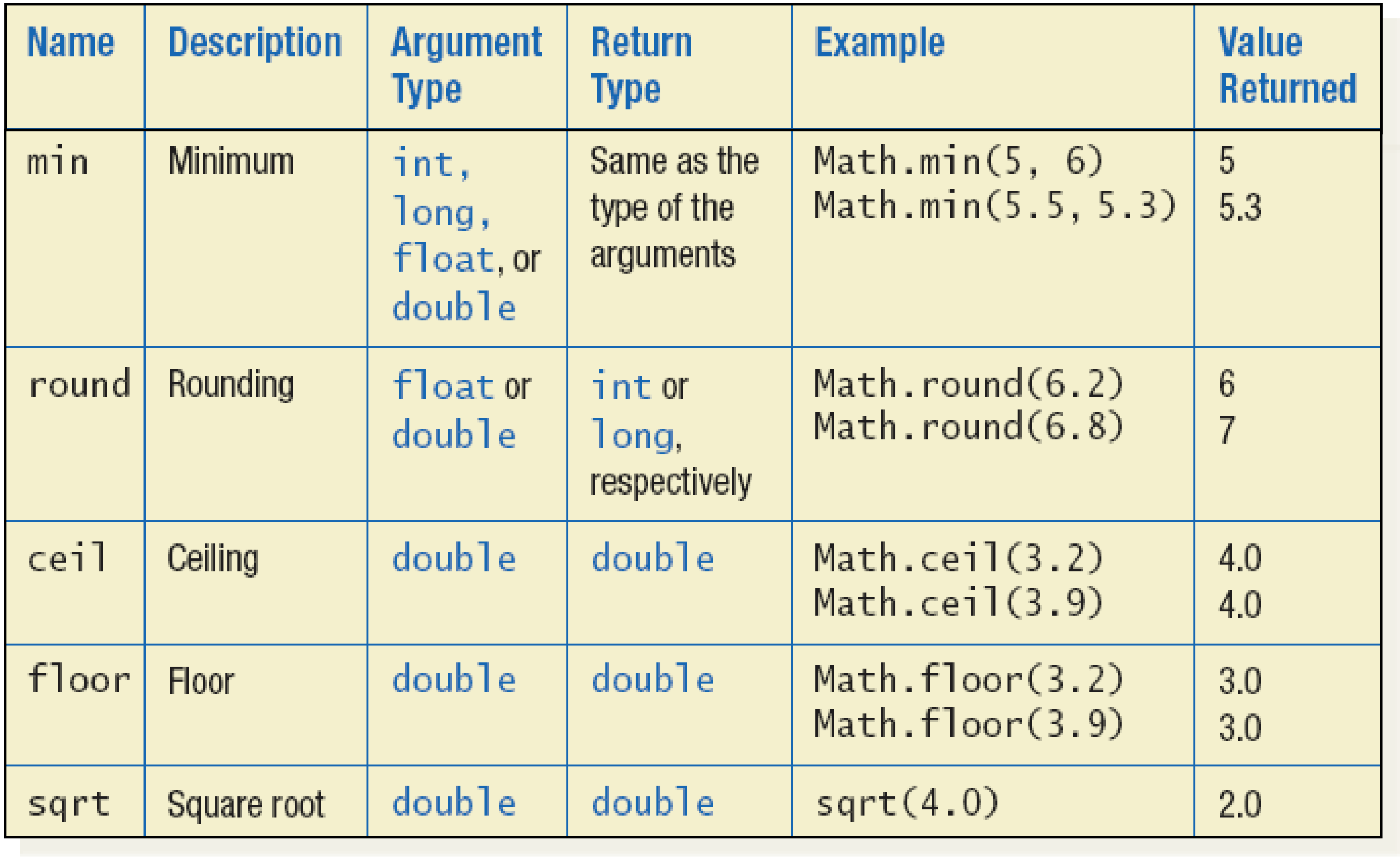
Name	Description	Argument Type	Return Type	Example	Value Returned
pow	Power	double	double	Math.pow(2.0,3.0)	8.0

abs	Absolute value	int, long, float, or double	Same as the type of the argument	Math.abs(-7)
				Math.abs(7)
				Math.abs(-3.5)
					7
					7
					3.5

max	Maximum	int, long, float, or double	Same as the type of the arguments	Math.max(5, 6)
				Math.max(5.5, 5.3)
					6
					5.5

Math Class

- Example methods, figure 6.3b



Name	Description	Argument Type	Return Type	Example	Value Returned

min	Minimum	int, long, float, or double	Same as the type of the arguments	Math.min(5, 6)
				Math.min(5.5, 5.3)
					5
					5.3

round	Rounding	float or double	int or long, respectively	Math.round(6.2)
				Math.round(6.8)
					6
					7

ceil	Ceiling	double	double	Math.ceil(3.2)
				Math.ceil(3.9)
					4.0
					4.0

floor	Floor	double	double	Math.floor(3.2)
				Math.floor(3.9)
					3.0
					3.0

sqrt	Square root	double	double	sqrt(4.0)	2.0

Random Numbers

- **Math.random()** returns a random double that is greater than or equal to zero and less than 1
- Java also has a **Random** class to generate random numbers
- Can scale using addition and multiplication; the following simulates rolling a six sided die

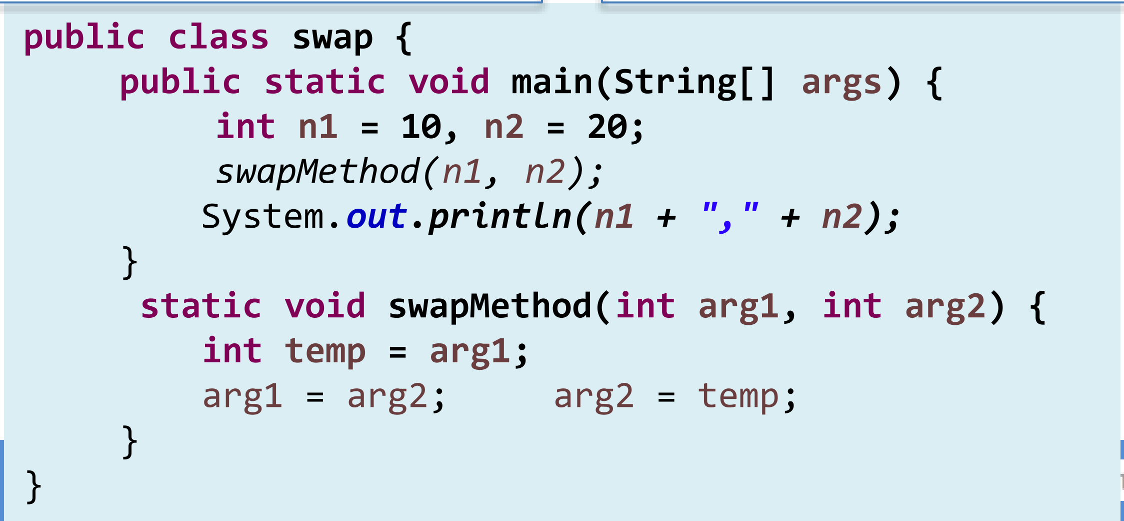
```
    int die = (int) (6.0 * Math.random()) + 1;
```

Question

- Can you write a traditional swap(a,b) method ?
  - Swap(a,b) : switching the values of variables a and b

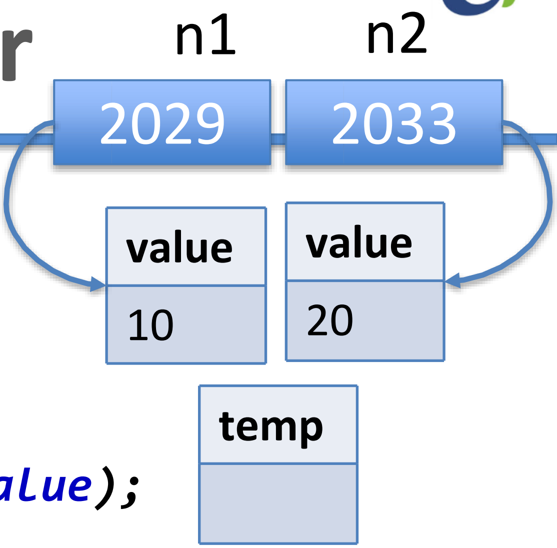
```
void swap(int arg1, int arg2) {
    int temp = arg1;
    arg1 = arg2;
    arg2 = temp;
}
```

```
main( ) {
    int n1 = 10, n2 = 20;
    swap(n1, n2);
    System.out.println(n1+”,”+n2);
}
```



```
public class swap {
     public static void main(String[] args) {
          int n1 = 10, n2 = 20;
          swapMethod(n1, n2);
         System.out.println(n1 + "," + n2);
     }
      static void swapMethod(int arg1, int arg2) {
         int temp = arg1;
         arg1 = arg2;     arg2 = temp;
     }
}
```

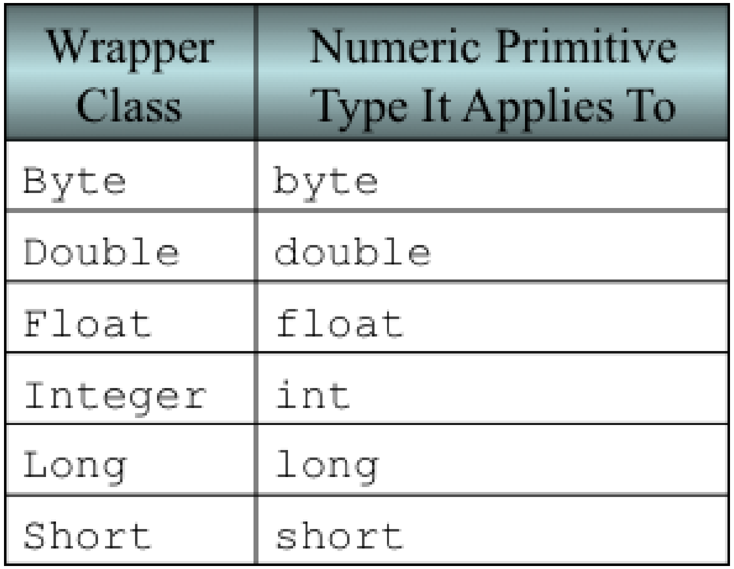
Possible solution – int holder



```
public class test {
    public static void main(String[] args) {
        IntHolder n1 = new IntHolder();
        IntHolder n2 = new IntHolder();
        n1.value = 10; n2.value =20;
        System.out.println(n1.value + " " + n2.value);
        swap(n1, n2);
        System.out.println(n1.value + " "  + n2.value);
    }
    static void swap(IntHolder a, IntHolder b){
        int temp = a.value;
        a.value = b.value;
        b.value = temp;
    }
}
class IntHolder { public int value = 0; }
```

Wrapper Classes

- Recall that arguments of primitive type treated differently from those of a class type
  - May need to treat primitive value as an object
- Java provides *wrapper classes* for each primitive type
  - **`Byte`**, **`Short`**, **`Long`**, **`Float`**, **`Double`**, and **`Character`**



Wrapper Class	Numeric Primitive Type It Applies To
`Byte`	`byte`
`Double`	`double`
`Float`	`float`
`Integer`	`int`
`Long`	`long`
`Short`	`short`

Wrapper Classes

- *Boxing*: the process of going from a value of a primitive type to an object of its wrapper class

```
Integer obj = new Integer(42);
Integer obj = Integer.valueOf(10000);
```

- *Unboxing*: the process of going from an object of a wrapper class to the corresponding value of a primitive type

```
int i = obj.intValue();
char a = obj.charValue();
```

To Boxing : valueOf()
To unboxing: type+Value()

Creating a Wrapper Class : Boxing

- To create objects from these wrapper classes, you can pass a value to the constructor:

```
    Integer number = new
    Integer(7);
```

Boxing

  - Wrapper classes have no default constructor : Programmer must specify an initializing value when creating new object

- You can also assign a primitive value to a wrapper class object:

```
    Integer number  = 7;
```

Automatic Boxing

  - Starting with JDK 1.5, Java can automatically do boxing and unboxing
  - No need to creat a wrapper class object using the **`new`** operation

Constants in Wrapper Classes

- Constants that provide the largest and smallest values for any of the primitive number types
  - Integer.MAX_VALUE, Integer.MIN_VALUE, Double.MAX_VALUE, Double.MIN_VALUE

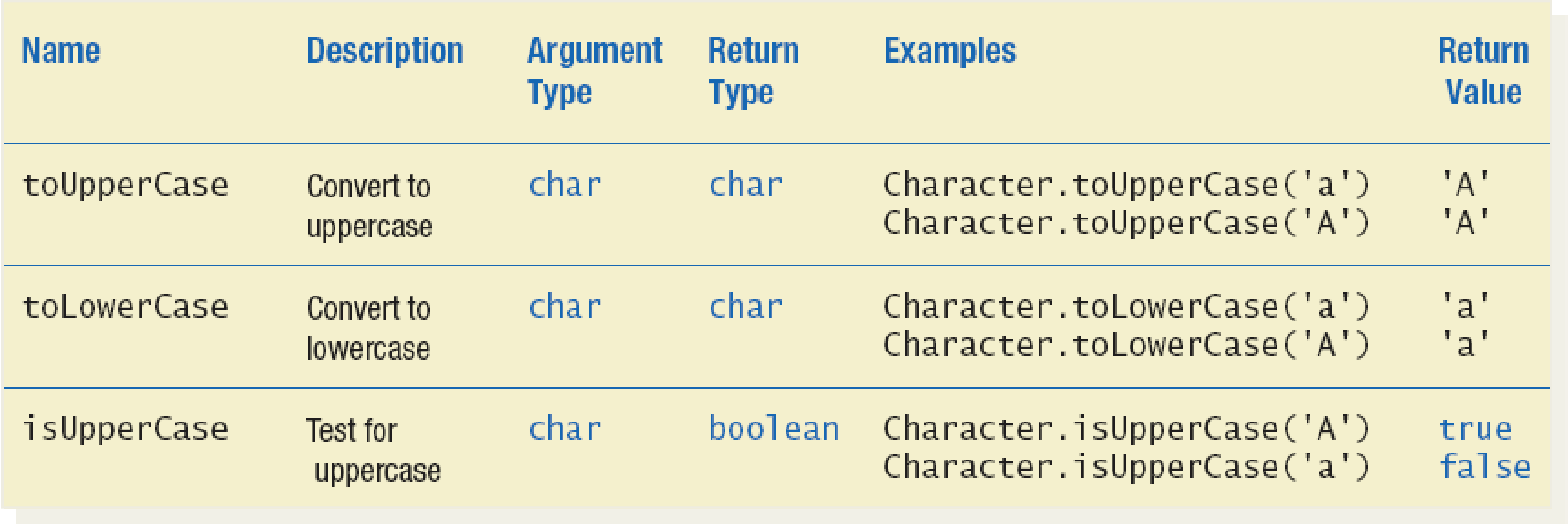
- Constants of type Boolean in Boolean class
  - Boolean.TRUE and Boolean.FALSE
  - Boolean objects that correspond to the values true and false of the primitive type boolean

Method in wrapper class

- valueOf(), valueOf(String s), valueOf(String s, int radix)
  - To create Wrapper object for given primitive or String.
- xxxValue()
  - To get the primitive for the given Wrapper Object
- parseXxx(), parseXxx(String s, int radix)
  - To convert String to primitive
- toString(), toString(, int radix)
  - Every wrapper class contains the following toString() method to convert Wrapper Object to String type.

Wrapper Classes

- Figure 6.4a Static methods in class `Character`



Name	Description	Argument Type	Return Type	Examples	Return Value

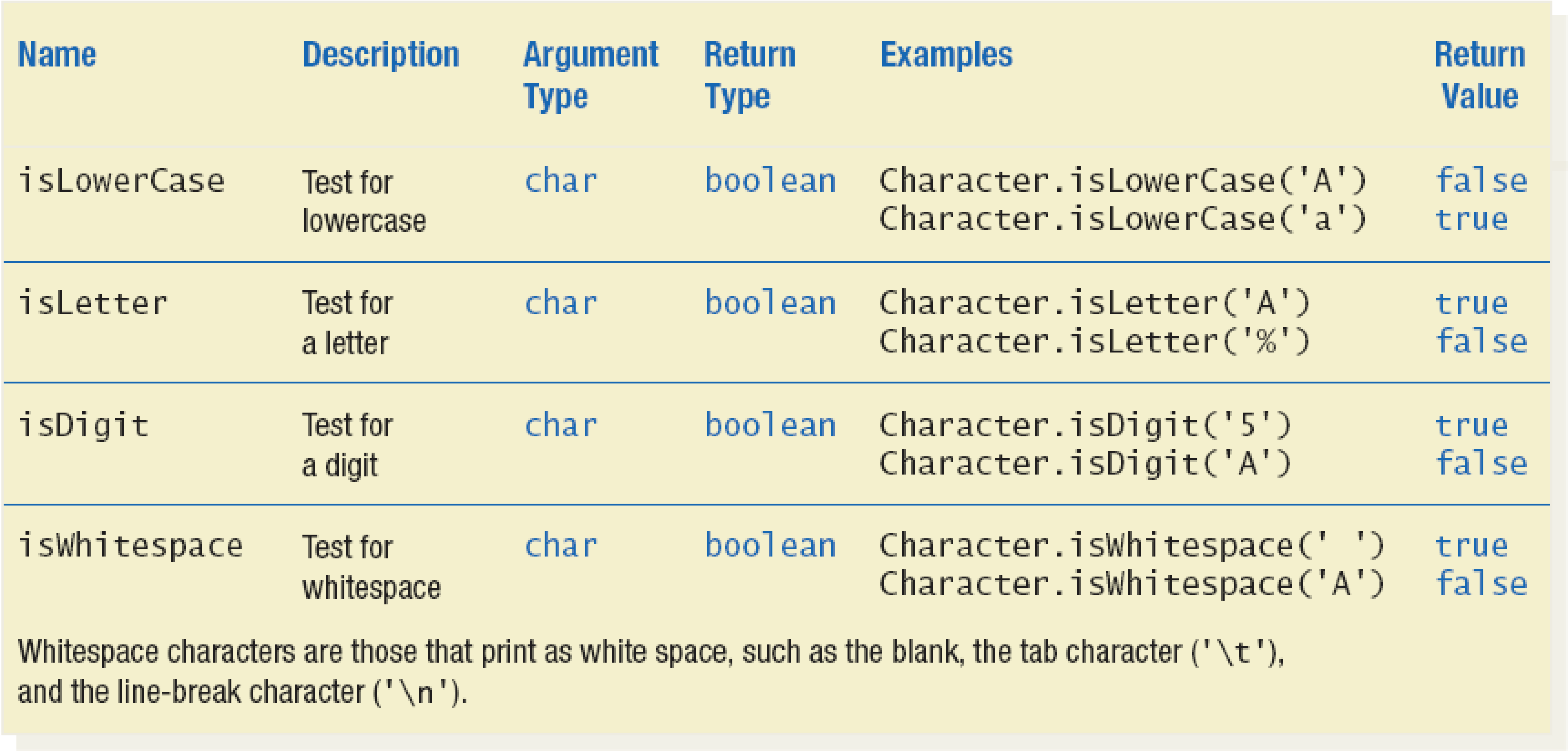
`toUpperCase`	Convert to uppercase	`char`	`char`	`Character.toUpperCase('a')`
				`Character.toUpperCase('A')`
					`'A'`
					`'A'`

`toLowerCase`	Convert to lowercase	`char`	`char`	`Character.toLowerCase('a')`
				`Character.toLowerCase('A')`
					`'a'`
					`'a'`

`isUpperCase`	Test for uppercase	`char`	`boolean`	`Character.isUpperCase('A')`
				`Character.isUpperCase('a')`
					`true`
					`false`

Wrapper Classes

- Figure 6.4b Static methods in class **Character**



Name	Description	Argument Type	Return Type	Examples	Return Value

`isLowerCase`	Test for lowercase	`char`	`boolean`	`Character.isLowerCase('A')`
				`Character.isLowerCase('a')`
					`false`
					`true`

`isLetter`	Test for a letter	`char`	`boolean`	`Character.isLetter('A')`
				`Character.isLetter('%')`
					`true`
					`false`

`isDigit`	Test for a digit	`char`	`boolean`	`Character.isDigit('5')`
				`Character.isDigit('A')`
					`true`
					`false`

`isWhitespace`	Test for whitespace	`char`	`boolean`	`Character.isWhitespace(' ')`
				`Character.isWhitespace('A')`
					`true`
					`false`

Whitespace characters are those that print as white space, such as the blank, the tab character (`'\t'`), and the line-break character (`'\n'`).

Lab: Wrapper class

```java
public class wrapper {
public static void main(String[] args) {
     Double avg = new  Double("3.145");
     System.out.println(avg);

     Integer I = Integer.valueOf("10");
     System.out.println(I);
     Double D = Double.valueOf("10.0");
     System.out.println(D);
     Boolean B = Boolean.valueOf("true");
     System.out.println(B);
     Character C = Character.valueOf('a');
     System.out.println(C);

     Integer I1 = Integer.valueOf("1111", 2);
     System.out.println(I1);

     System.out.println(I.byteValue());
     System.out.println(I.shortValue());
     System.out.println(I.intValue());
     System.out.println(I.longValue());
     System.out.println(I.floatValue());
     System.out.println(I.doubleValue());

     int i = Integer.parseInt("10");
     double d = Double.parseDouble("10.5");
     boolean b = Boolean.parseBoolean("true");
     System.out.println(i);
     System.out.println(d);
     System.out.println(b);
```

```java
     String s = I.toString();
     System.out.println(s);
     String s1 = Integer.toString(15, 2);
     System.out.println(s1);

     Integer age1 = new Integer(30);
     Integer age2 = 30; // age2와 3에 1000 테스트
     Integer age3 = 30;


     System.out.println(age1);
     System.out.println(age1.intValue());

     System.out.println(age2);
     System.out.println(age2.intValue());

     if(age1 == age2)
     System.out.println("reference same");

     if(age2 == age3)
     System.out.println("reference same");

     if(age1.intValue() == age2.intValue())
     System.out.println("value same");
     }
}
```

Practice 6.1

- Ex6_1. Create a class RoomOccupancy
  - Instance variables (private): roomNumber, peopleInRoom
  - Static variables: totalNumber – number of total people
  - Write the following methods:
    - Constructor – gets roomNumber and peopleInRoom parameters
    - addOneToRoom – increases peopleInRoom and totalNumber
    - removeOneFromRoom – decreases peopleInRoom and totalNumber, if both are > 0
    - getNumber – returns roomNumber
    - getTotal (static) – returns totalNumber
  - Write a program to test the class with ≥2 objects

Practice 6.1

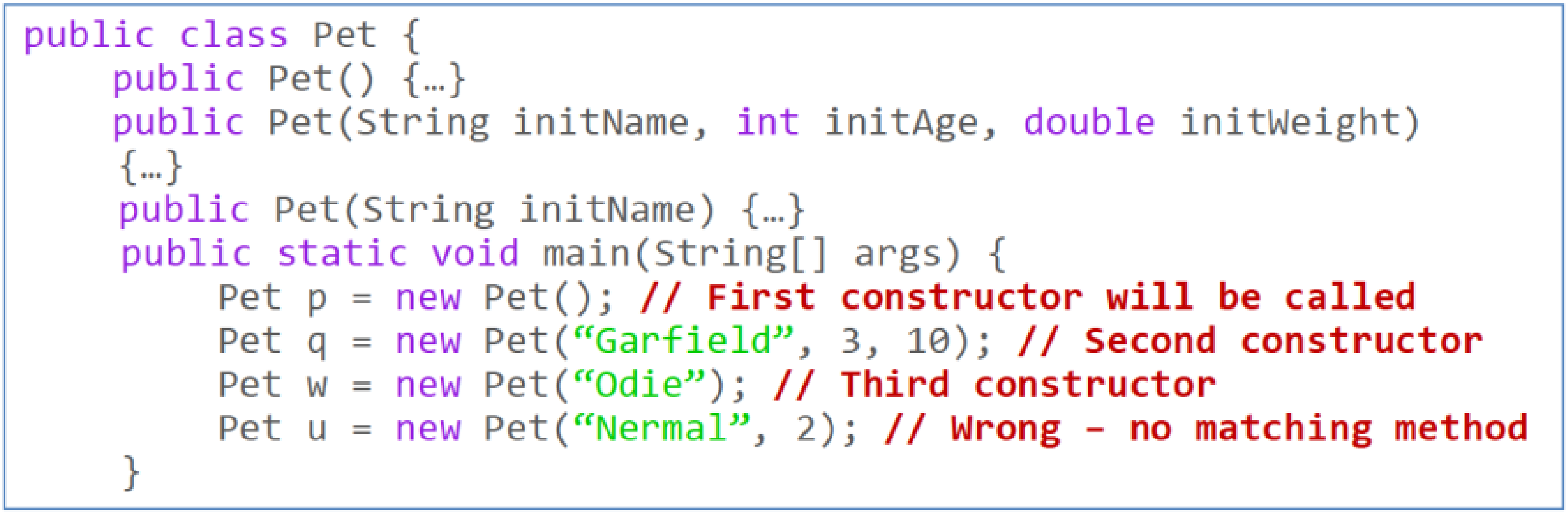
- Test class

```java
public class RoomOccupancyDemo {
    public static void main(String[] args) {
        RoomOccupancy r = new RoomOccupancy(101, 2);
        RoomOccupancy s = new RoomOccupancy(102, 3);
        System.out.println(RoomOccupancy.getTotal());

        r.addOneToRoom();
        r.addOneToRoom();
        s.removeOneFromRoom();

        System.out.println(r.getPeopleInRoom());
        System.out.println(s.getPeopleInRoom());
        System.out.println(RoomOccupancy.getTotal());
    }
}
```

6.4 Overloading

Methods Overloading

- We’ve seen that a class can have multiple constructors. Notice that they have the same name.



```
public class Pet {
    public Pet() {…}
    public Pet(String initName, int initAge, double initWeight)
     {…}
     public Pet(String initName) {…}
     public static void main(String[] args) {
          Pet p = new Pet(); // First constructor will be called
          Pet q = new Pet(“Garfield”, 3, 10); // Second constructor
          Pet w = new Pet(“Odie”); // Third constructor
          Pet u = new Pet(“Nermal”, 2); // Wrong – no matching method
     }
```

Overloading Basics

- When two or more methods have same name within the same class
  - *It is not only for constructors*
- Parameter lists **must** be different
  - Java distinguishes the methods by number and types of parameters; it attempts to do type conversions
  - `public double average(int n1, int n2)`
  - `public double average(double n1, double n2)`
  - `public double average(double n1, double n2, double n3)`

Overloading : Method Signature

- A method's **name** and **number** and **type** of parameters is called the ***signature***
  - Java knows what to use based on the number and types of the arguments

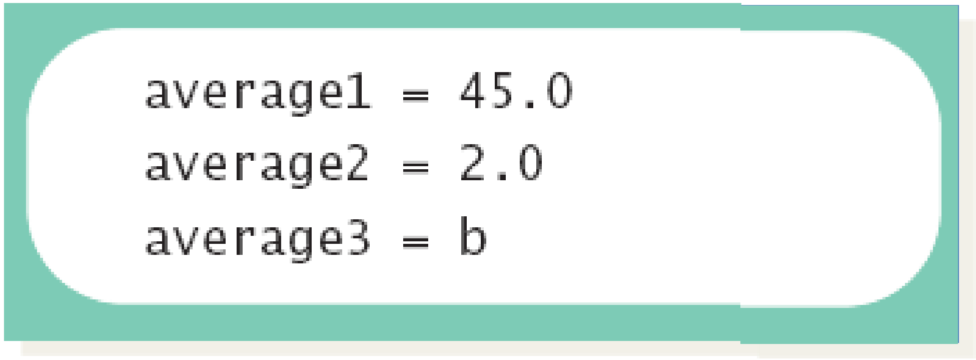
```
System.out.println(“The result is”); // String type parameter
System.out.println(20); // int type parameter
```

- **Signature does NOT include return type**
  - Cannot have two methods with the same signature in the same class

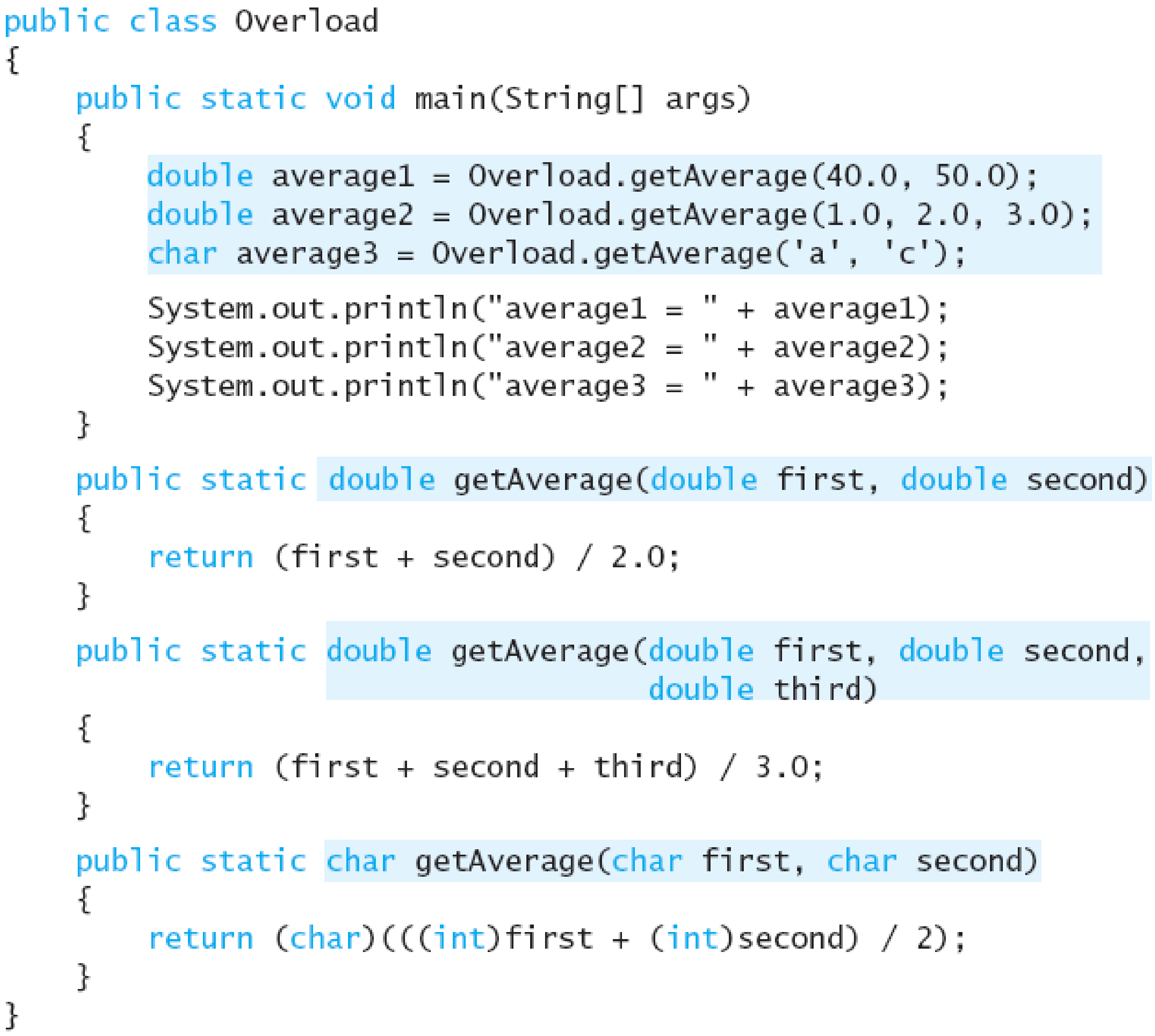
```
public double average(int n1, int n2)
public int average(int n1, int n2)   // Wrong overloading
public int average(char n1, char n2) // it is okay
```

Overloading Basics

- View example program, listing 6.15 class Overload
- Note overloaded method getAverage



```
average1 = 45.0
average2 = 2.0
average3 = b
```



```
public class Overload
{
    public static void main(String[] args)
    {
        double average1 = Overload.getAverage(40.0, 50.0);
        double average2 = Overload.getAverage(1.0, 2.0, 3.0);
        char average3 = Overload.getAverage('a', 'c');
        System.out.println("average1 = " + average1);
        System.out.println("average2 = " + average2);
        System.out.println("average3 = " + average3);
    }
    public static double getAverage(double first, double second)
    {
        return (first + second) / 2.0;
    }
    public static double getAverage(double first, double second,
                                    double third)
    {
        return (first + second + third) / 3.0;
    }
    public static char getAverage(char first, char second)
    {
        return (char)(((int)first + (int)second) / 2);
    }
}
```

Overloading and Type Conversion

- Java always tries to find an exactly matching method. If it fails, it tries type conversion
- If a class has the following two methods:
  - `public double average(int n1, int n2)`
  - `public double average(double n1, double n2)`
    - If the method call is average(3,3), the first method will be called
  - However, if a class only have a method:
  - `public double average(double n1, double n2)`
    - If the method call is average(3,3), it will be converted to average(3.0,3.0) and call the (only) method
  - Recall: byte->short->int->long->float->double

How to Use Overloading

- Use it only if two or more methods are performing exactly the same function
  - `public void setPet(String newName)`
  - `public void setPet(String newName, int newAge, double newWeight)`
- Bad idea to create methods that have the same name but do different things
  - `public void setPet(int newAge)`
  - `public void setPet(double newWeight)`
  - What happens if we call setPet(3)?
    - Use setAge() and setWeight() instead
    - Usually we do not overload methods if parameters can be converted

6.6 Enumeration as a Class

Enumeration

- Consider the case to restrict a variable to have only the values in a certain set
  - An *enumeration* lists the values a variable can have
  - Example
    - enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES}
    - Suit suit=Suit.DIAMONDS;

Enumeration as a Class

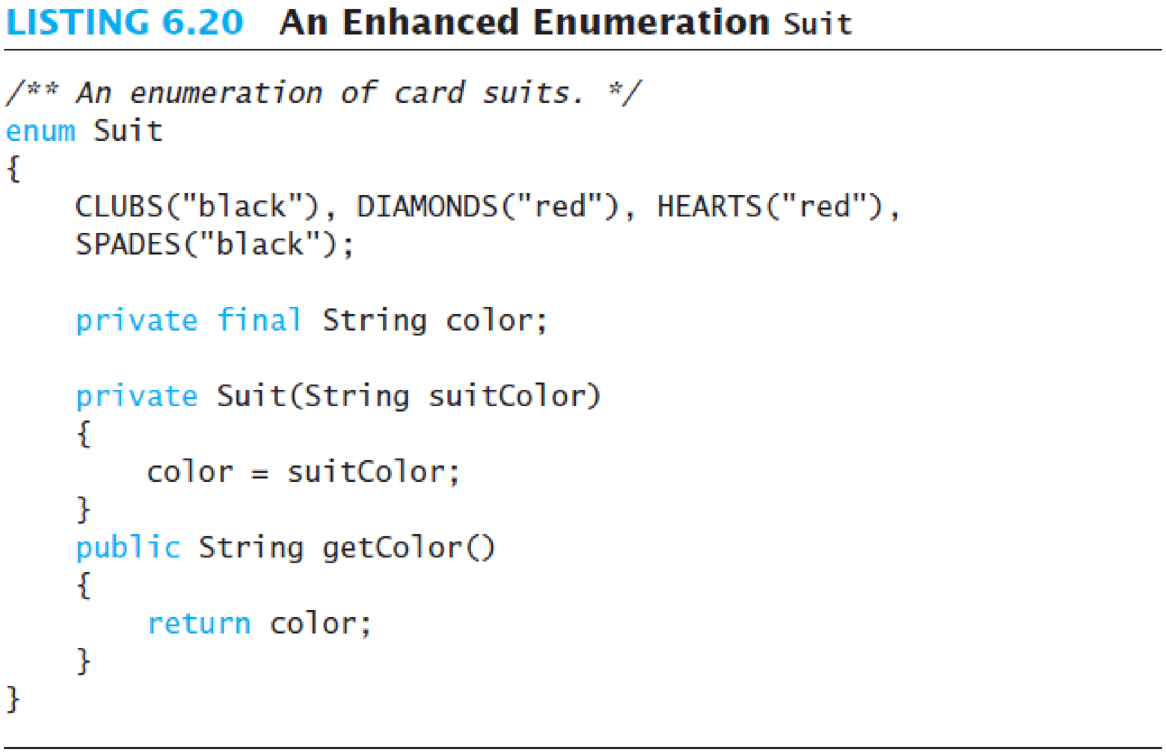
- When an enumeration is defined, Java creates a class with methods:
- Compiler creates a class with methods
  - equals() – returns true if equal
  - compareTo() – returns a negative value if comes earlier
  - ordinal() – returns the position ≥ 0
  - toString() – returns a String
  - valueOf() – returns an enum value

Enumeration as a Class

protected Object	clone()Throws CloneNotSupportedException.
int	compareTo(E o)Compares this enum with the specified object for order.
boolean	equals(Object other)Returns true if the specified object is equal to this enum constant.
protected void	finalize()enum classes cannot have finalize methods.
Class<E>	getDeclaringClass()Returns the Class object corresponding to this enum constant's enum type.
int	hashCode()Returns a hash code for this enum constant.
String	name()Returns the name of this enum constant, exactly as declared in its enum declaration.
int	ordinal()Returns the ordinal of this enumeration constant (its position in its enum declaration, where the initial constant is assigned an ordinal of zero).
String	toString()Returns the name of this enum constant, as contained in the declaration.
static <T extends Enum<T>> T	valueOf(Class<T> enumType, String name)Returns the enum constant of the specified enum type with the specified name.

Enhanced enumeration

- Define your own class for enumeration – with additional instance variables and methods (including constructors)



LISTING 6.20 **An Enhanced Enumeration** `Suit`

```
/** An enumeration of card suits. */
enum Suit
{
    CLUBS("black"), DIAMONDS("red"), HEARTS("red"),
    SPADES("black");

    private final String color;

    private Suit(String suitColor)
    {
        color = suitColor;
    }
    public String getColor()
    {
        return color;
    }
}
```

Lab: enum test

- Write a developer class that use enum type for development type
- Use enum class functions to get data

```
enum DevType1 { MOBILE, WEB, SERVER }
enum DevType2 {
    MOBILE("Android"), WEB("CSS"), SERVER("LINUX");
    final private String name;
    public String getName() {      return name;      }
    private DevType2(String name){   this.name = name;     }
}
```

```java
public class test {
public String name;
    public int career;
    public DevType1 type1;

    public static void main(String[] args) {
    test developer = new test();

    // test basic enum usage
    developer.name = "Kim";
    developer.career = 3;
    developer.type1 = DevType1.WEB;
    System.out.println("Developer name : "+ developer.name);
    System.out.println("Experience : "+ developer.career);
    System.out.println("Experties : "+ developer.type1);

    // test enum class methods 1
    DevType1 tp1 = DevType1.MOBILE;
    DevType1 tp2 = DevType1.valueOf("WEB");
    System.out.println(tp1);
    System.out.println(tp2);
    System.out.println(tp1.ordinal());
    System.out.println(tp2.ordinal());
    for(DevType1 type1 : DevType1.values()){
            System.out.println(type1);
    }

    // test enum class methods 2
    for( DevType2 type2: DevType2.values()){
            System.out.println(type2.getName());
    }
  }
}
```

```java
// 기존 for 구문
String[] numbers = {"one", "two", "three"};
   for(int i=0; i<numbers.length; i++)
      { System.out.println(numbers[i]); }
```

```java
// For each 구문
for (type var: iterate)
    { body-of-loop }

String[] numbers = {"one", "two", "three"};
for(String number: numbers)
    { System.out.println(number); }
```

6.7 Packages

Packages and Importing

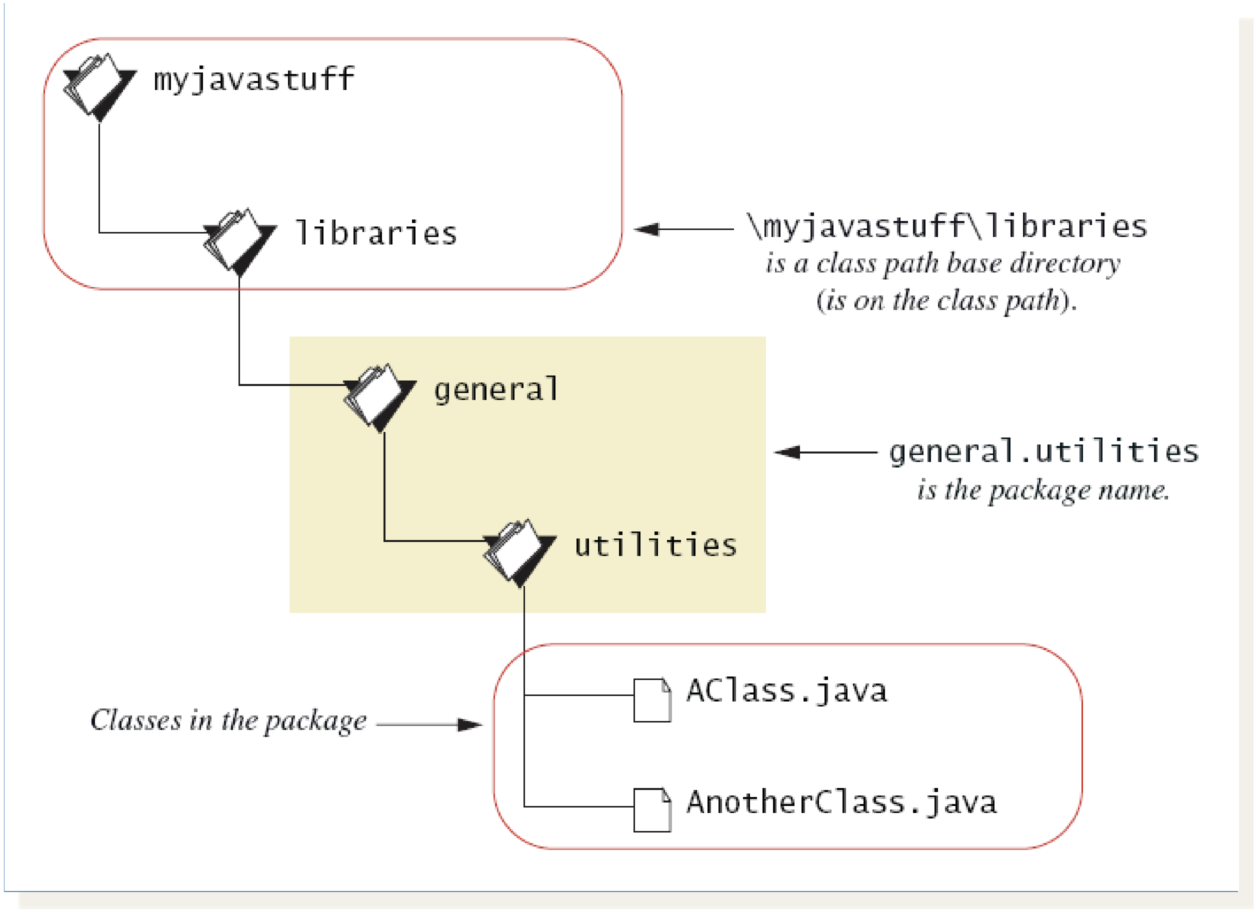
- A package is a collection of classes grouped together into a folder
- Name of folder is name of package
- Each class
  - Placed in a separate file
  - Has this line at the beginning of the file
    package Package_Name;
- Classes use packages by use of import statement

Package Names and Directories

- Package name tells compiler path name for directory containing classes of package
- Search for package begins in class path base directory
  - Package name uses dots in place of / or \
- Name of package uses relative path name starting from any directory in class path

Package Names and Directories

- Figure 6.5 A package name



Name Clashes

- Packages help in dealing with name clashes
  - Different programmers may give same name to two classes
  - Ambiguity resolved by using the package name

```
<folder>
src/species/cat/kitty.java
src/species/dog/poodle/puppy.java
```

How to define?

```
Package species.cat;
public class kitty { }

Package species.dog.poodle;
public class puppy { }
```

How to use?

```
import species.cat.kitty;
import species.cat.*;
import species.dog.poodle.puppy;
import species.dog.poodle.*;
```