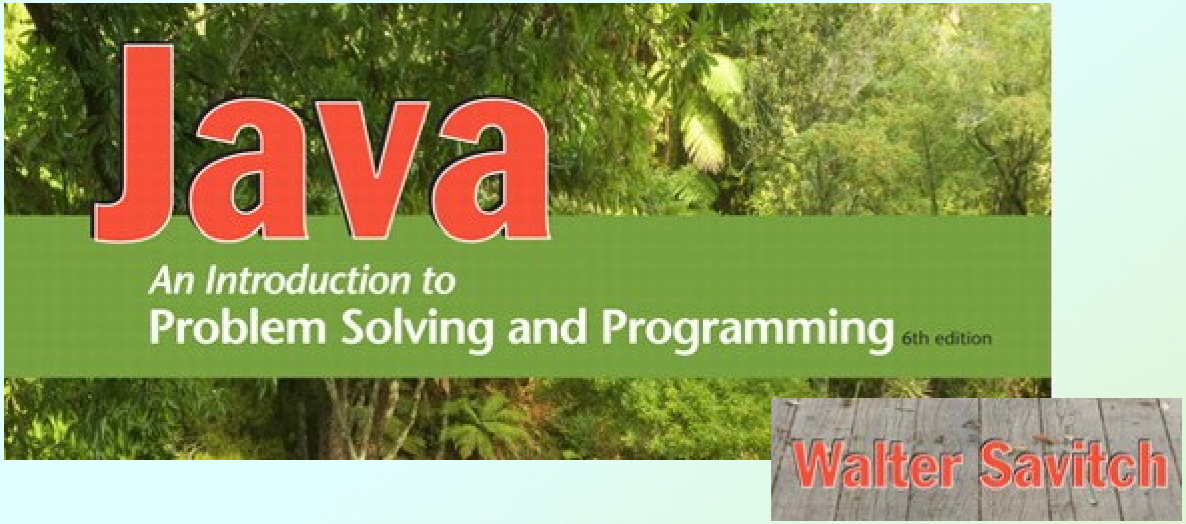


More About Objects and Methods

Chapter 6

Objectives

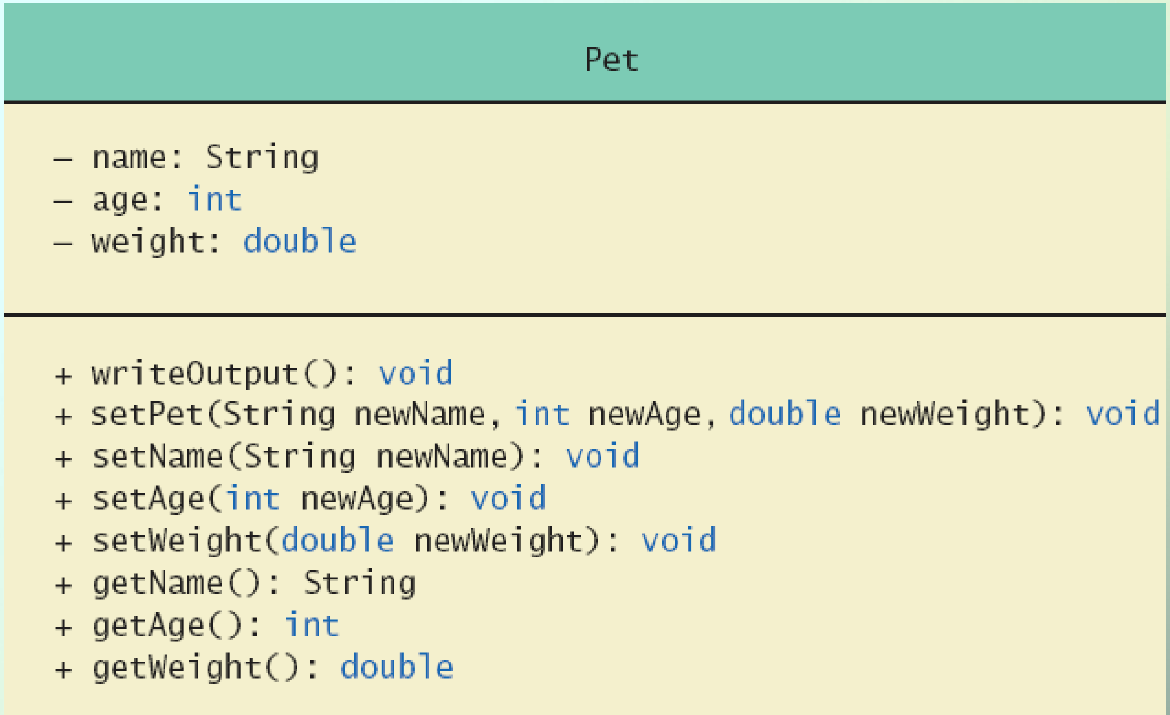
- Define and use constructors
- Write and use static variables and methods
- Use methods from class **`Math`**
- Use predefined wrapper classes
- Write and use overloaded methods
- Use **`import`** statements

Defining Constructors

- A special method called when instance of an object created with **new**
  - Create objects
  - Initialize values of instance variables
- Can have parameters
  - To specify initial values if desired
- May have multiple definitions
  - Each with different numbers or types of parameters

Defining Constructors

- Example class to represent pets
- Figure 6.1 Class Diagram for Class **`Pet`**

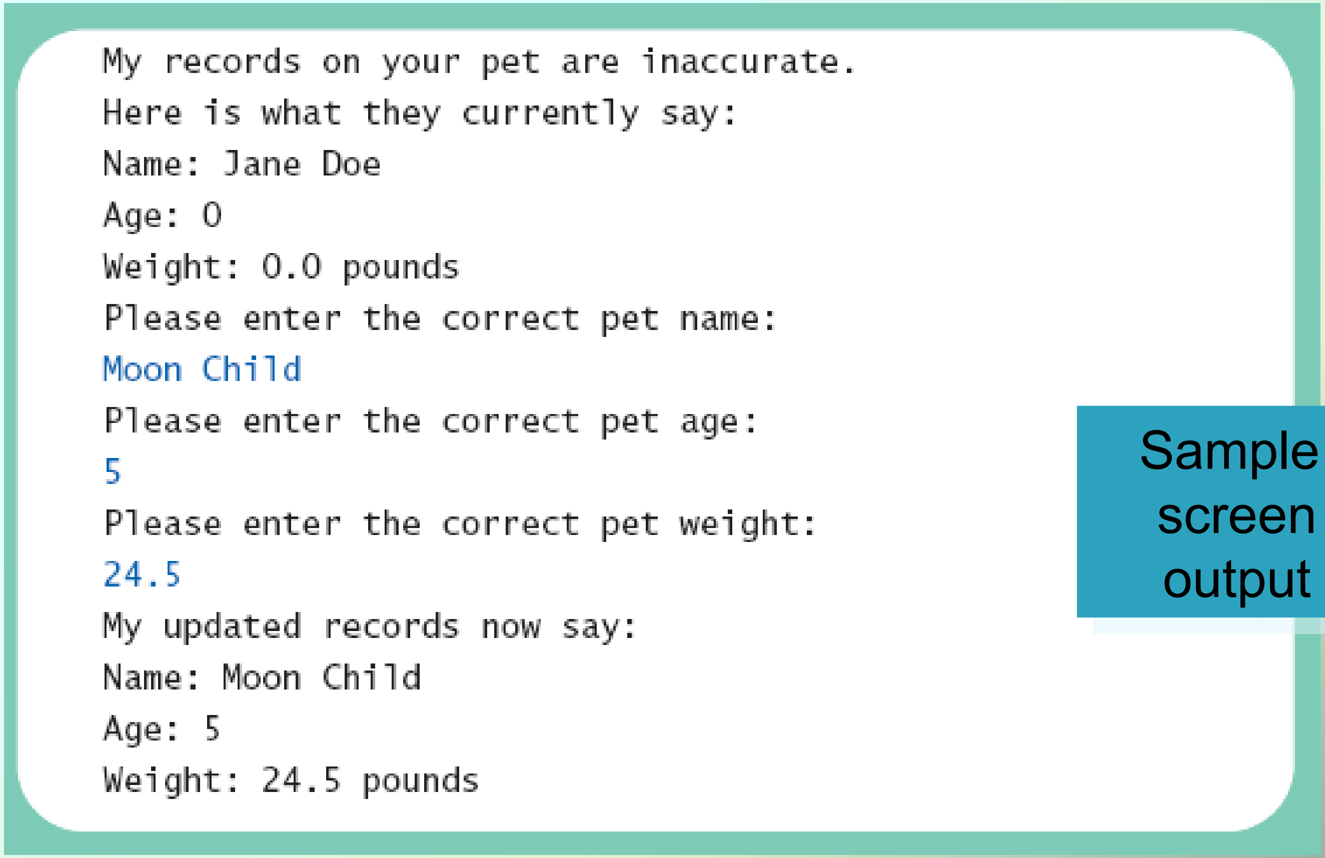


```
Pet

– 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
```

Defining Constructors

- Download `Pet.java` and `PetDemo.java`
- Note different constructors
  - Default
  - With String parameter
  - With int parameter
  - With double parameter
  - With 3 parameters

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
```

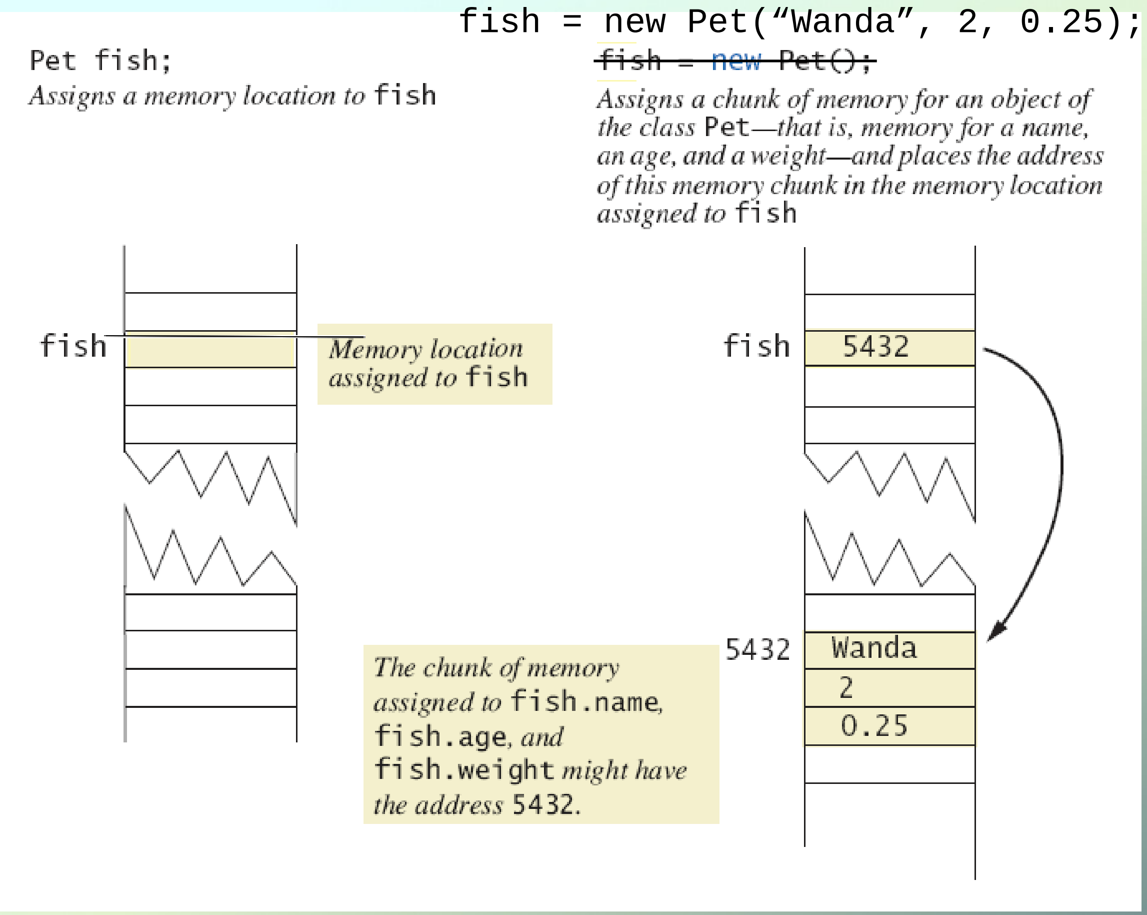
Sample screen output

Defining Constructors

- Constructor without parameters is the default constructor
  - 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
- Constructors not always included in UML class diagram

Defining Constructors

- Figure 6.2 A constructor returning a reference



```
fish = new Pet(“Wanda”, 2, 0.25);
```

~~fish = new Pet();~~

Pet fish;

Assigns a memory location to fish

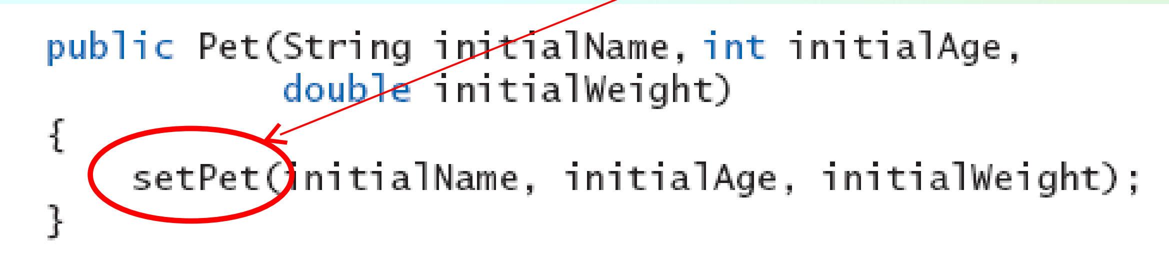
Assigns a chunk of memory for an object of the class Pet*—that is, memory for a name, an age, and a weight—and places the address of this memory chunk in the memory location assigned to* fish

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

- Constructor can call other class methods



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

- Change all of the **`Pet`** constructors to call the 3-parameter **`set`** method (see **`Pet2.java`**)
  - Error handling in one place in **`set`** method

Static Variables

- Static variables are shared by all objects of a class
  - Variables declared `static final` are considered constants – value cannot be changed
- Variables declared `static` (without `final`) can be changed
  - Only one instance of the variable exists
  - It can be accessed by all instances of the class

Static Variables

- Static variables also called *class variables*
  - Shared by all instances of the class
  - Contrast with *instance variables* (each object has its own instance variables)
- Do not confuse class variables with variables of a class type
- Both static variables and instance variables are sometimes called *fields* or *data members*

Modify `BankAcct`

- Add an instance variable `int acctNum` to your `BankAcct` class
  - Include a `getAcctNum` method
  - Update your `toString` method

Modify `BankAcct`

- Add an instance variable `int acctNum` to your `BankAcct` class
  - Include a `getAcctNum` method
  - Update your `toString` method
- Add 2 constructors to your `BankAcct` class
  - default constructor (sets balance to 0)
  - a constructor that takes the initial balance as a parameter and sets balance to that

Modify `BankAcct`

- We want each `BankAcct` to get a unique account number
- We can do this with a static variable
- Add a static variable `nextAcctNum` to your `BankAcct` class:

```
private static int nextAcctNum = 0;
```

Modify **BankAcct**

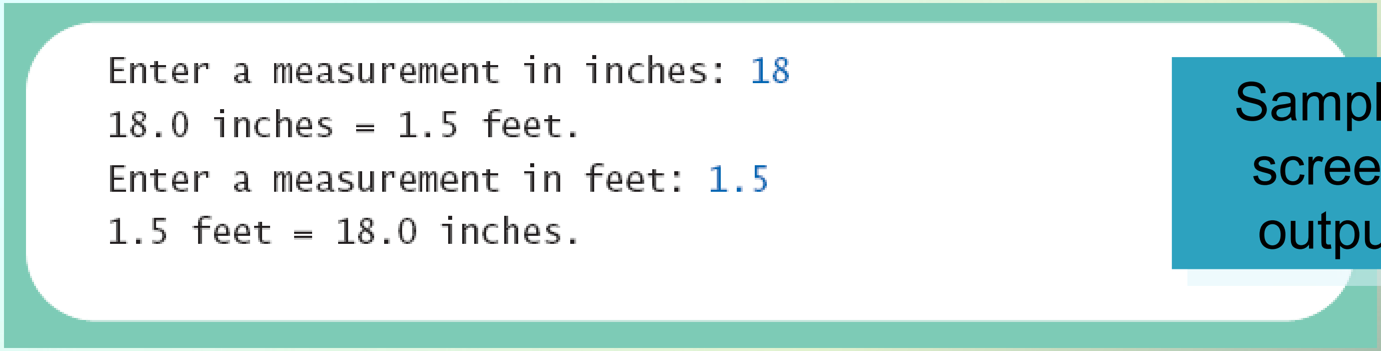
- Add code in each constructor so that each time an account is created:
  - the account number is set to the next account number
  - the next account number is incremented
- Create at least 3 accounts in your demo program and print them

Static Methods

- Some methods may have no relation to any type of object
- Examples:
  - Compute max of two integers
  - Convert character from upper- to lower case
- Static method declared in a class
  - Can be invoked without using an object
  - Instead use the class name
- Example: `int absValue = Math.abs(8 - 12);`

Static Methods

- Download **DimensionConverter** and **DimensionConverterDemo**

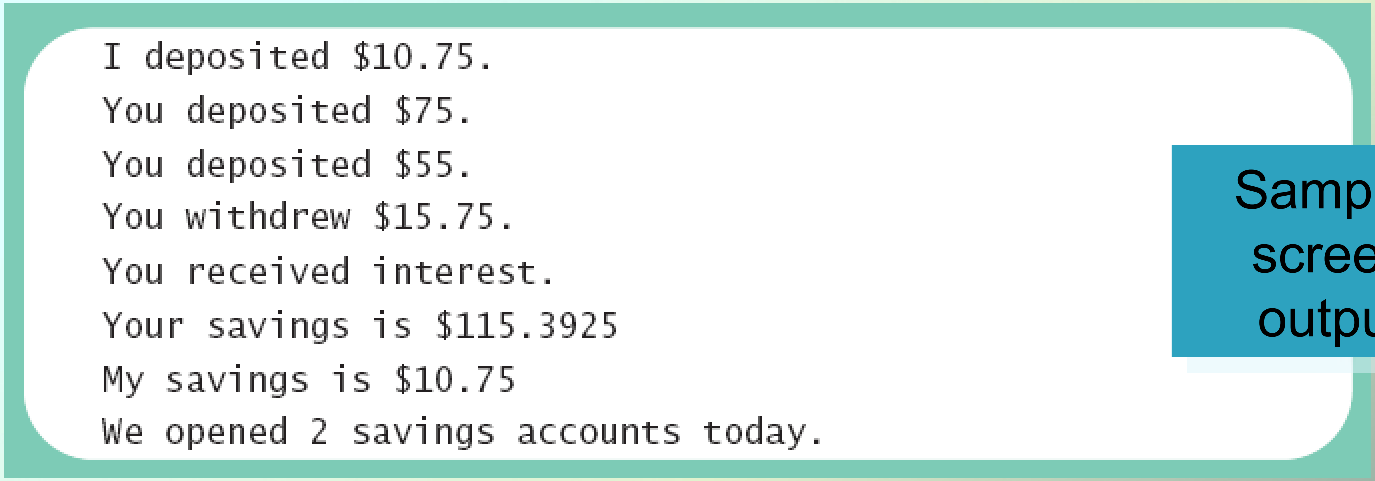


```
Enter a measurement in inches: 18
18.0 inches = 1.5 feet.
Enter a measurement in feet: 1.5
1.5 feet = 18.0 inches.
```

Sample screen output

Mixing Static and Nonstatic Methods

- Download **`SavingsAccount`** and **`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.
```

Sample screen output

Static Methods

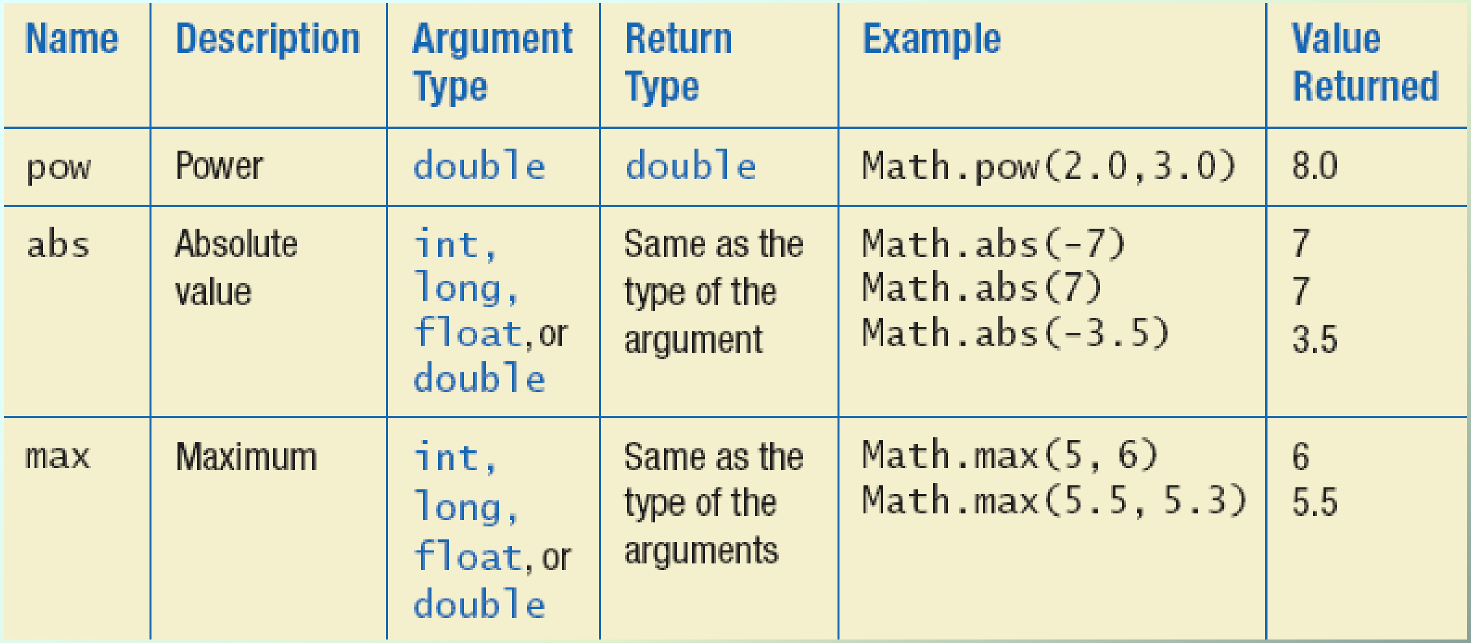
- Static methods are not allowed to access instance variables
- Static methods can only call other static methods in the class definition
- *main* is a static method
- All of the methods in the **`Math`** class in the java library are static
- See the **`Math`** docs in the Java API

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
- Download ch06/`Species`
  - When used as ordinary class, method `main` ignored

The `Math` Class

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



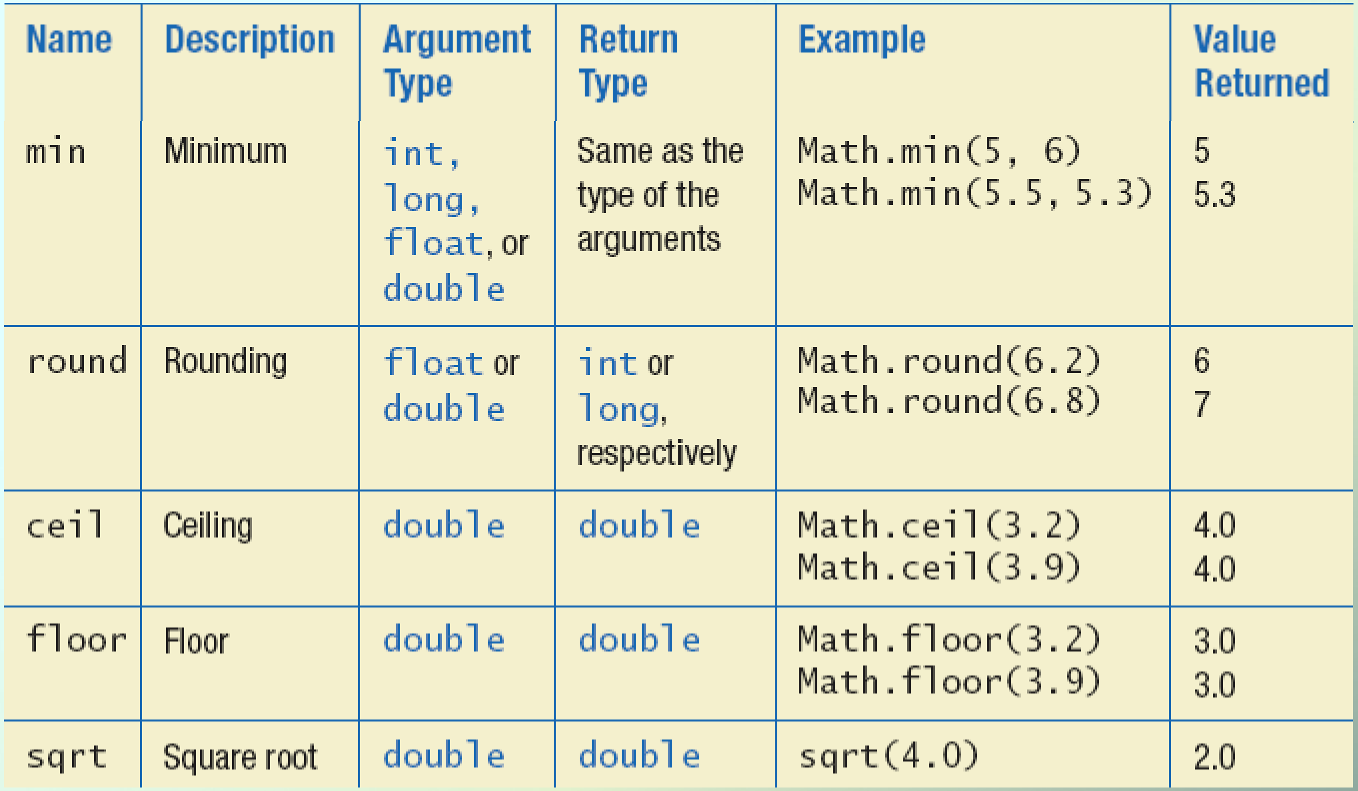
Name	Description	Argument Type	Return Type	Example	Value Returned
`pow`	Power	`double`	`double`	`Math.pow(2.0,3.0)`	8.0

`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

The 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;
```

Exercise

- Add a static method to your **BankAcct** class that transfers a positive amount from one account to another (it's ok to have 2 methods with the same name – more on that soon):

```
public static void transfer(BankAcct from, BankAcct to,
                            double amount)
{

}
```

Wrapper Classes

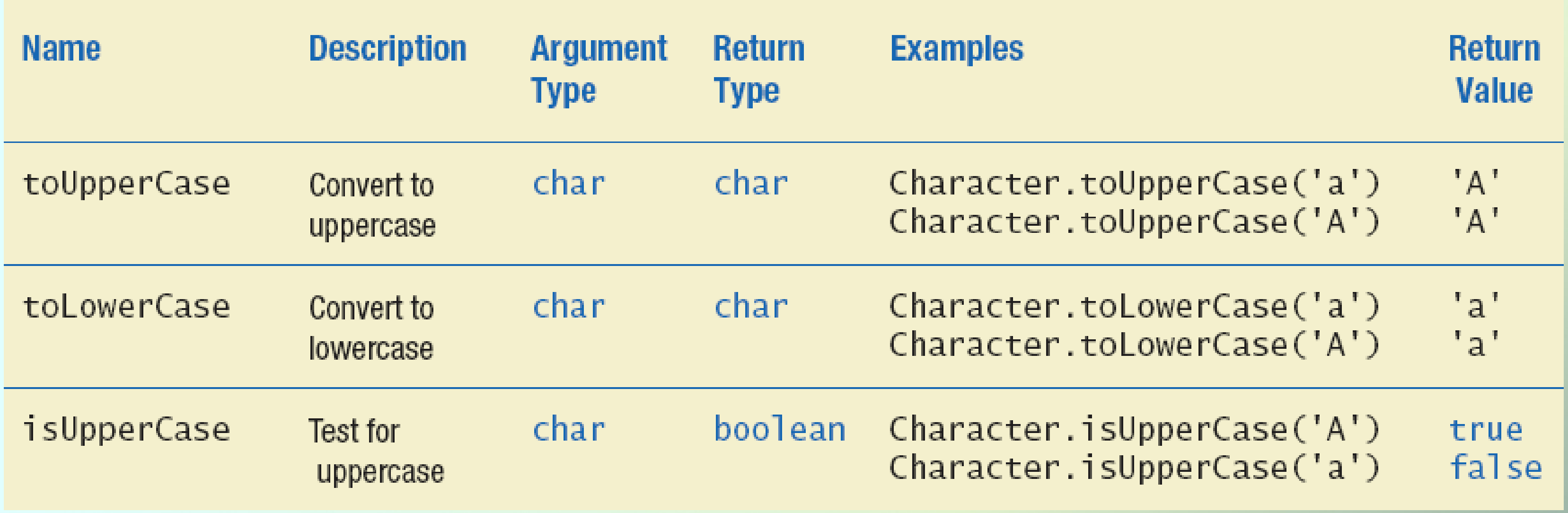
- Recall that arguments of primitive type treated differently from those of a class type
    - May need to treat primitive value as an object
- Java provides *wrapper classes* for each primitive type
    - Methods provided to act on values

Wrapper Classes

- Allow programmer to have an object that corresponds to value of primitive type
- Contain useful predefined constants and methods
- Wrapper classes have no default constructor
  - Programmer must specify an initializing value when creating new object
- Wrapper classes have no `set` methods

Wrapper Classes

- Figure 6.4a Static methods in class `Character`



Name	Description	Argument Type	Return Type	Examples	Return Value

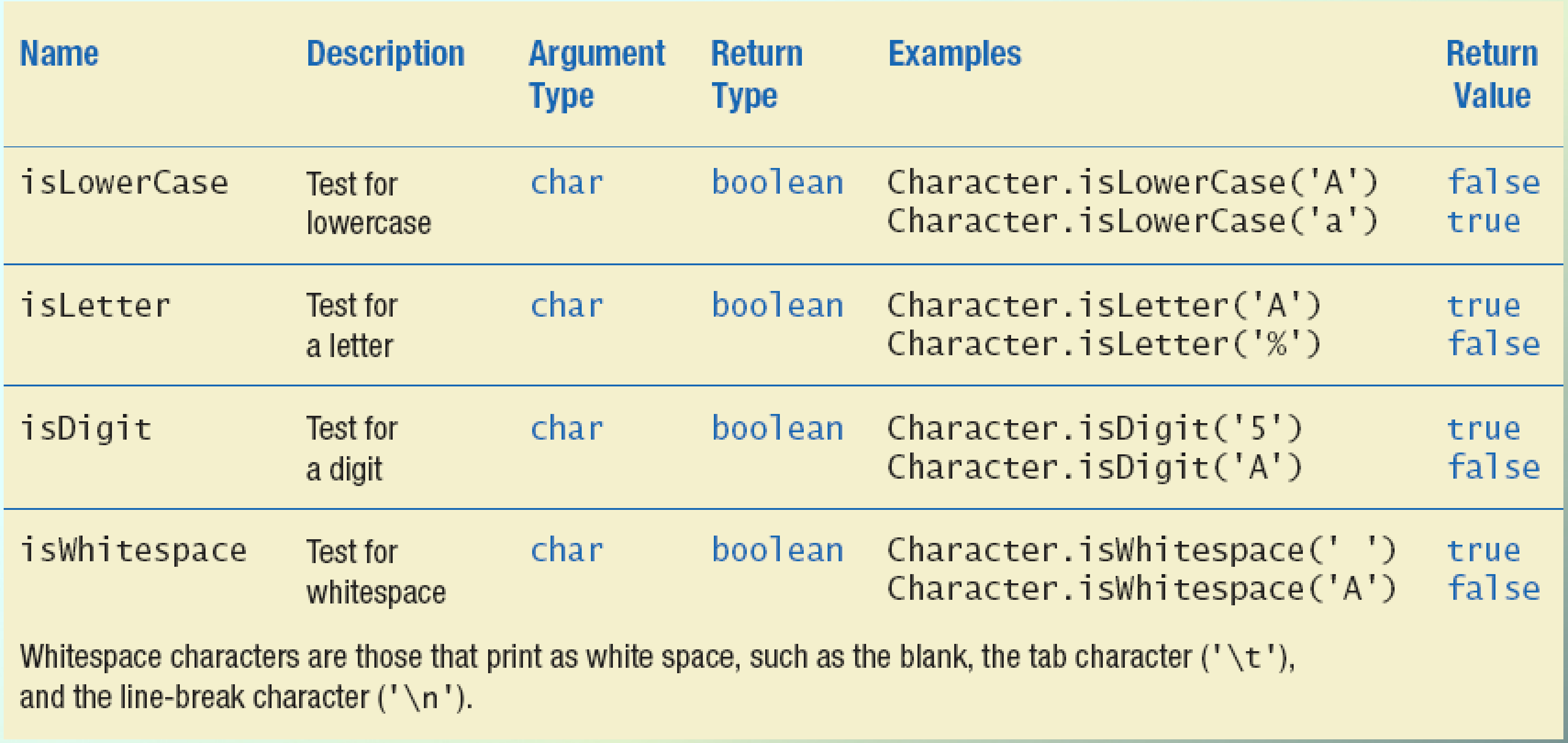
`toUpperCase`	Convert to uppercase	`char`	`char`	`Character.toUpperCase('a')`
				`Character.toUpperCase('A')`
					`'A'`
					`'A'`

`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'`).

Overloading Basics

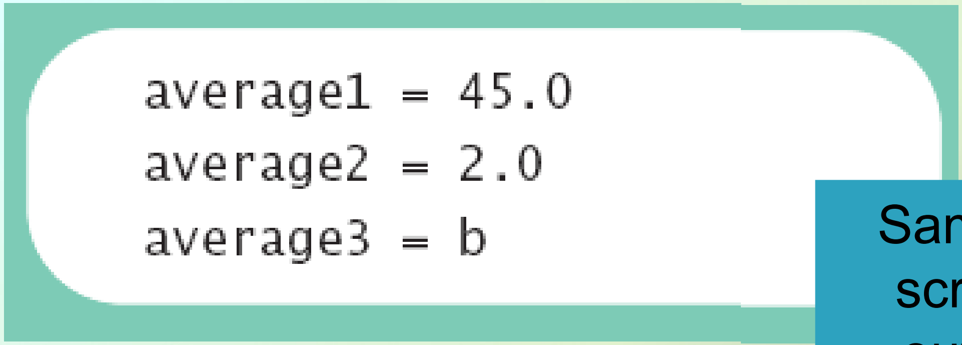
- When two or more methods have same name within the same class it is called *overloading*
- Java distinguishes the methods by number and types of parameters
  - If it cannot match a call with a definition, it attempts to do type conversions
- A method's name and number and type of parameters is called the *signature*

Overloading Basics

- We have been using overloaded methods all along
- In the `String` class:
  - `myString.substring(3);`
  - `myString.substring(0, 5);`
- In the `PrintStream` class:
  - `System.out.println(42);`
  - `System.out.println(“Hello”);`

Overloading Basics

- Download **`Overload.java`**
- Note overloaded method **`getAverage`**



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

Sample screen output

Overloading Basics

- Overloaded constructors or methods must have
  - different number of parameters:

```
public String substring(int startIndex) {
public String substring(int startIndex, int endIndex) {
```

  - OR different types of parameters

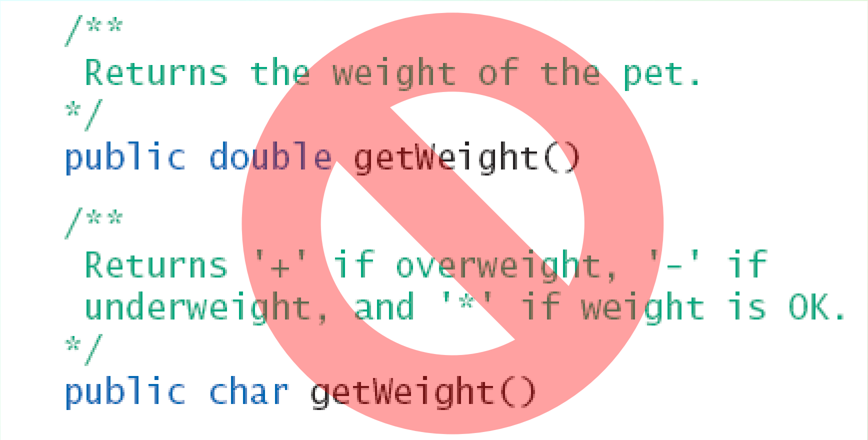
```
public void println(int x) {
public void println(double d) {
public void println(boolean b) {
public void println(String s) {
```

Overloading and Type Conversion

- Overloading and automatic type conversion can conflict
- Recall definition of `Pet` class
  - If we pass an integer to the constructor we get the constructor for age, even if we intended the constructor for weight
  - Would be better not to include contructors that take age and weight only
- Remember the compiler only does type conversion if an exact match is not found

Overloading and Return Type

- You must not overload a method where the only difference is the type of value returned



```
/**
 Returns the weight of the pet.
*/
public double getWeight()

/**
 Returns '+' if overweight, '-' if
 underweight, and '*' if weight is OK.
*/
public char getWeight()
```