

1

Method Overloading

Overloading Basics

2

- A method's **formal parameter list** is the number and types of its parameters
- A method's name **and** formal parameter list is called the method's **signature**
- Two methods are said to have **different** formal parameter lists if they have:
 - ▣ a different number of parameters,
OR
 - ▣ the data type of the formal parameters, in the order listed, differ in at least one position

Overloading Basics

3

- When two or more methods have **same name** within the **same class** they are said to be **overloaded**:
 - ▣ they **must** have different signatures though,
 - ▣ since the name is the same it means the formal parameter list **must** be different.
- Java distinguishes these methods by number and types of parameters
 - ▣ If it cannot match a call with a definition, it attempts to do type conversions

If a method name is overloaded, then the formal parameter list determines which method to execute when called.

Method Formal Parameter List

4

```
public void methodOne (int x)
public void methodTwo (int x, double y)
public void methodThree(double x, int y)
public int  methodFour (char ch, int x, double y)
public int  methodFive (char ch, int x, String name)
```

- These methods all have different formal parameter lists

Method Formal Parameter List

5

```
public void methodOne (int )
public void methodTwo (int , double )
public void methodThree(double , int )
public int  methodFour (char , int , double )
public int  methodFive (char , int , String )
```

- These methods all have **different** formal parameter lists
- The names of the parameters are irrelevant, only the type is relevant

Method Formal Parameter List

6

```
public void methodSix (int x, double y, char ch)
public void methodSeven (int one, double u, char firstCh)
```

- The methods `methodSix` and `methodSeven` both have three formal parameters and the data type of the corresponding parameters is the same
- These methods all have the **same** formal parameter lists

Method Formal Parameter List

7


```
public void methodSix (int , double , char )  
public void methodSeven (int , double , char )
```

- ❑ The methods `methodSix` and `methodSeven` both have three formal parameters and the data type of the corresponding parameters is the same
- ❑ These methods all have the **same** formal parameter lists
- ❑ Remember: parameter name is irrelevant

Method Overloading

8

- ❑ **Method overloading:** creating one or more methods with **the same** name and **different formal parameter lists** within a **class**,
- ❑ **The signature of a method:**
 - ▣ Consists of the method name and its formal parameter list
 - ▣ It does **not** include the return type of the method
- ❑ Two methods have **different signatures** if they have:
 - ▣ either different names or different formal parameter lists



the signature does NOT include the return type

Method Overloading (continued)

9

- The following method headings **correctly** overload the method `methodXYZ`:

```
public void methodXYZ ()
```

```
public void methodXYZ (int x, double y)
```

```
public void methodXYZ (double one, int y)
```

```
public void methodXYZ (int x, double y, char ch)
```

- What about these? Why?

```
public int methodXYZ (int xx, double yy) ❌
```

```
public char methodXYZ (double one, int y) ❌
```

```
public char methodXYZ (char one, char y) ✅
```

Method Overloading (continued)

10

```
public void methodABC (int x, double y)
public int  methodABC (int x, double y)
```

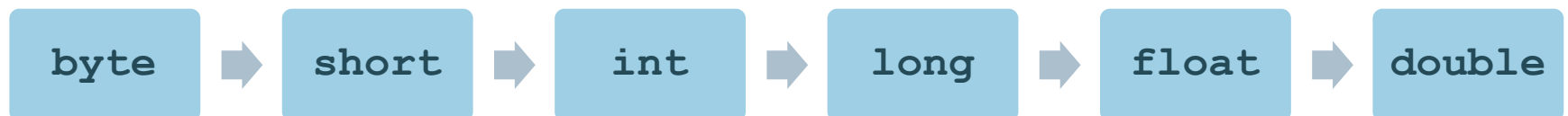
- Both these method headings have the same name and same formal parameter list
- These method headings are **incorrect** for the purpose of overloading the method `methodABC`
- The compiler will generate a syntax error
 - Notice that the return types of these method headings are different

Overloading and Type Conversion

- Remember the compiler attempts to overload **before** it does type conversion
 - ▣ for example: if you have

```
public void over(double x)
public void over(byte x)
public void over(long y)
```
 - ▣ calling:

```
over(3)
```
 - ▣ will have the compiler **first** look for an overloaded method over that takes an **integer**, if it does not find it, it will try to do conversion, here it converts to **long**



Overloading and Type Conversion

- ❑ Overloading and automatic type conversion can conflict

- ❑ for example: if you have

```
public void over(double x, double y)      // d d
public void over(double x, int y)         // d int
public void over(int x, double y)         // int d
```

- ❑ calling:

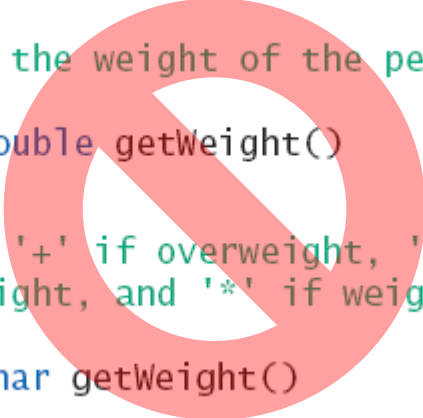
```
over(3, 3);
```

- ❑ will have the compiler **first** look for an overloaded method over that takes two integers, if it does not find it, it will try to do conversion
 - ❑ here it gets confusing → ambiguous → causes an error because the 2nd and the 3rd version could be matched.

Overloading and Return Type

- ❑ You must not overload a method where the **only** difference is the type of value returned
- ❑ return value is NOT part of the signature

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



EXAMPLES

5.1 int/long/double/float abs(int/long/double/float x)

- Recall the pre-defined method `abs`:
- In fact, it is written four times with different formal parameters:

- `public static int abs (int x)`
- `public static long abs (long x)`
- `public static double abs (double x)`
- `public static float abs (float x)`

- In other words, the method name `abs` is overloaded.

If a method name is overloaded, then the formal parameter list determines which method to execute when called. Type conversion is the 2nd step.

The 3rd way is a double, stored in an int

The 1st is called (takes an int), result is an int, CAN be stored in a double

```
1 int num1 = 10;  
2 double num2 = 50.23;  
3 num1 = abs(num1); //The first method is called (int)  
4 num2 = abs(num2); //The third method is called (double)  
  
5 num1 = abs(num2); //valid? which method is called?  
6 num2 = abs(num1); //valid? Which method is called?
```

5. EXAMPLES

The method `larger` is overloaded with different parameter lists.

```
17 public static int larger (int num1, int num2) {
18     if (num1 > num2)
19         return num1;
20     return num2; } // end of larger (int, int)

21 public static char larger (char ch1, char ch2) {
22     if (ch1 > ch2)
23         return ch1;
24     return ch2; } // end of larger (ch, ch)

25 public static double larger (double num1, double num2)
26 {
27     if (num1 > num2)
28         return num1;
29     return num2;
30 } // end of larger (double, double)
```

5. EXAMPLES

- We can even define more methods with three or more parameters:

```
31 public static int larger (int num1, int num2, int num3) {  
32     int max = num1;  
32     if (num2 > max)  
33         max = num2;  
34     if (num3 > max)  
35         max = num3;  
36     return max;  
37 } // end of larger (int num1, int num2, int num3)  
    } //end class
```

- Similarly, more parameters may be defined with more overloaded methods.
- However, a method should be defined for every different formal parameter list (different type and/or different number).
- Java provides the concept of **variable length parameter list** to simplify rewriting the method multiple times in the above example...

EXAMPLES

Consider this main method calling `larger` with different actual parameters.

```
1 public class overloading
2 { public static void main (String[] args)
3   { int resultInt2, resultInt3;
4     char resultChar;
5     double resultDouble;
6     resultInt2 = larger (5, 9); // calls line 17
7     resultInt3 = larger (40, -20, 3); // calls line 31
8     resultChar = larger ('A', 'Z'); // calls line 21
9     resultDouble = larger (55.5, 30.2); //calls line 25
10    System.out.printf ("resultInt2 = %d\n", resultInt2);
11    System.out.printf ("resultInt3 = %d\n", resultInt3);
12    System.out.printf ("resultChar = %c\n", resultChar);
13    System.out.printf ("resultDouble = %f\n",
14                       resultDouble);
15 } //end main
```