METHODS REITERATION

OUTLINE

1. PASSING DATA AMONG METHODS
2. OVERLOADING METHODS

1) PASSING DATA AMONG METHODS

- **Java is a “pass-by-value” language.**
- **This means that a copy of the variable is made and the method receives that copy.**
- **Assignments made in the method do not affect the caller.**

```
4⊖    public static void main(String[] args) {
5         int num = 4;
6         newNumber(num);
7         System.out.println(num); // 4
8     }
9
10⊖   public static void newNumber(int num) {
11        num = 8;
12    }
```

- On line 5, `num` is assigned the value of 4.
- On line 6, we call a method.
- On line 11, the `num` parameter in the method gets set to 8.
- Although this parameter has the same name as the variable on line 5, **this is a coincidence**. The name could be anything.

1) PASSING DATA AMONG METHODS

- Now that you've seen primitives, let's try an example with a reference type.

```
4⊖    public static void main(String[] args) {
5         StringBuilder name = new StringBuilder("John");
6         method(name);
7         System.out.println(name); //John
8     }
9
10⊖   public static void method(StringBuilder sb) {
11        sb = new StringBuilder("Dave");
```

- The output will be John.
- Just as in the primitive example, **the variable assignment is only to the method parameter and doesn't affect the caller.**

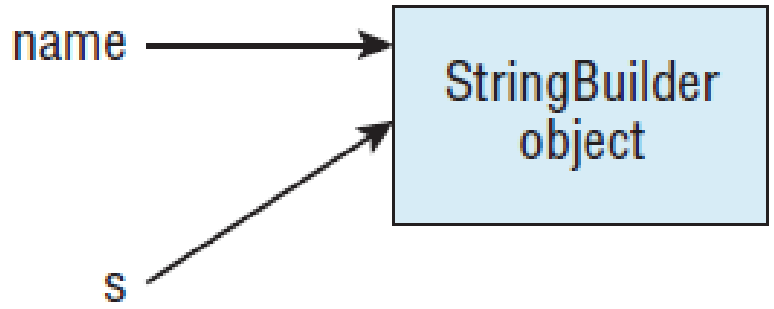
1) PASSING DATA AMONG METHODS

- Notice how we keep talking about variable assignments.
- This is because we can call methods on the parameters.
- As an example, we have code that calls a method on the StringBuilder passed into the method:

```
4    public static void main(String[] args) {
5        StringBuilder name = new StringBuilder("John");
6        method(name);
7        System.out.println(name); //JohnDave
8    }
9
10   public static void method(StringBuilder s) {
11       s.append("Dave");
```

- In this case, the output is JohnDave because the method calls a method on the parameter.
- **It doesn't reassign name to a different object.**

- In the below figure, we can see how pass-by-value is used.



- s is a copy of the variable name.
- Both point to the same StringBuilder, which means that changes made to the StringBuilder are available to both references.

1) PASSING DATA AMONG METHODS

- To review, Java uses pass-by-value to get data into a method.
- Assigning a new primitive or reference to a parameter doesn't change the caller.
- Calling methods on a reference to an object does affect the caller.
- Getting data back from a method is easier. A copy is made of the primitive or reference and returned from the method.
- Most of the time, this returned value is used. For example, it might be stored in a variable.
- **If the returned value is not used, the result is ignored.**

```
4  public static void main(String[] args) {
5      int nr = 1; // 1
6      String letters = "abc"; // abc
7      number(nr); // 1
8      letters = letters(letters); // abcd
9      System.out.println(nr + letters); // 1abcd
10 }
11
12 public static int number(int number) {
13     number++;
14     return number;
15 }
16
17 public static String letters(String letters) {
18     letters += "d";
19     return letters;
20 }
```

2) OVERLOADING METHODS

- Now that we are familiar with the rules for declaring methods, it is time to look at creating methods with the same signature in the same class.
- ***Method overloading*** **occurs when there are different method signatures with the same name but different type parameters.**
- We've been calling overloaded methods for a while. System.out.println and StringBuilder's append methods provide many overloaded versions so we can pass just about anything to them without having to think about it.
- In both of these examples, the only change was the type of the parameter.
- **Overloading also allows different numbers of parameters.**
- Everything other than the method signature can vary for overloaded methods.
- This means there can be different access modifiers, specifiers (like static), return types, and exception lists.

2) OVERLOADING METHODS

- These are all valid overloaded methods:

```
public void fly(int numMiles) { }
public void fly(short numFeet) { }
public boolean fly() { return false; }
void fly(int numMiles, short numFeet) { }
public void fly(short numFeet, int numMiles) throws Exception { }
```

- As we can see, we can overload by changing anything in the parameter list.
- We can have a different type, more types, or the same types in a different order.
- Also notice that the access modifier and exception list are irrelevant to overloading.

2) OVERLOADING METHODS

- Now let's look at an example that is not valid overloading:

```
public void fly(int numMiles) { }
public int fly(int numMiles) { } // DOES NOT COMPILE
```

- This method doesn't compile because it only differs from the original by return type.
- The parameter lists are the same so they are duplicate methods as far as Java is concerned.
- What about these two? Why does the second not compile?

```
public void fly(int numMiles) { }
public static void fly(int numMiles) { } // DOES NOT COMPILE
```

- Again, the parameter list is the same.
- The only difference is that one is an instance method and one is a static method.

2) OVERLOADING METHODS

- Calling overloaded methods is easy. We just write code and Java calls the right one.
- For example, look at these two methods:

```
public void fly(int numMiles) {
    System.out.println("int");
}

public void fly(short numFeet) {
    System.out.println("short");
}
```

- The call `fly((short) 1);` prints `short`.
- It looks for matching types and calls the appropriate method.
- Of course, it can be more complicated than this.

2) OVERLOADING METHODS

AUTOBOXING

- Each primitive type has a wrapper class, which is an object type that corresponds to the primitive. The table below lists all the wrapper classes along with the constructor for each.

PRIMITIVE TYPE	WRAPPER CLASS	EXAMPLE OF CONSTRUCTING
boolean	Boolean	new Boolean(true)
byte	Byte	new Byte((byte) 1)
short	Short	new Short((short) 1)
int	Integer	new Integer(1)
long	Long	new Long(1)
float	Float	new Float(1.0)
double	Double	new Double(1.0)
char	Character	new Character('c')

2) OVERLOADING METHODS

- Since Java 5, we can just type the primitive value and Java will convert it to the relevant wrapper class for us. This is called **autoboxing**.

```
public void fly(Integer numMiles) { }
```

- This means calling `fly(3);` will call the previous method as expected.
- However, what happens if we have both a primitive and an integer version ?

```
public void fly(int numMiles) { }
public void fly(Integer numMiles) { }
```

- Java will match the `int numMiles` version.
- **Java tries to use the most specific parameter list it can find.**
- When the primitive `int` version isn't present, it will autobox.
- However, when the primitive `int` version is provided, there is no reason for Java to do the extra work of autoboxing.

2) OVERLOADING METHODS

REFERENCE TYPES

- Given the rule about Java picking the most specific version of a method that it can, let's look at the code. What will it output ?
- The answer is `"string object".`
- The first call is a `String` and finds a direct match.
- There's no reason to use the `Object` version when there is a nice `String` parameter list just waiting to be called.
- The second call looks for an `int` parameter list.
- When it doesn't find one, it autoboxes to `Integer`.
- Since it still doesn't find a match, it goes to the `Object` one.

```
public class ReferenceTypes {

    public void fly(String s) {
        System.out.print("string ");
    }

    public void fly(Object o) {
        System.out.print("object ");
    }

    public static void main(String[] args) {
        ReferenceTypes r = new ReferenceTypes();
        r.fly("test");
        r.fly(56);
    }
```

2) OVERLOADING METHODS

PRIMITIVE TYPES

- Primitives work in a way similar to reference variables.
- Java tries to find the most specific matching overloaded method. What will the code output ?
- The answer is `int long`. The first call passes an `int` and sees an exact match.
- The second call passes a `long` and also sees an exact match.
- If we comment out the overloaded method with the `int` parameter list, the output becomes `long long`.
- Java has no problem calling a larger primitive.
- However, it will not do so unless a better match is not found.
- **Note that Java can only accept wider types**. An `int` can be passed to a method taking a long parameter.
- **Java will not automatically convert to a narrower type**. If you want to pass a `long` to a method taking an `int` parameter, you have to add a cast to explicitly say narrowing is okay.

```
public class Plane {
    public void fly(int i) {
        System.out.print("int ");
    }

    public void fly(long l) {
        System.out.print("long ");
    }

    public static void main(String[] args) {
        Plane p = new Plane();
        p.fly(123);
        p.fly(123L);
    }
```

2) OVERLOADING METHODS

- Order Java uses to choose the right overloaded method:

RULE	EXAMPLE FOR `glide(1,2)`
Exact match by type	String glide(int i, int j) {}
Larger primitive type	String glide(long i, long j) {}
Autoboxed type	String glide(Integer i, Integer j) {}
Varargs	String glide(int... nums) {}

- It prints out 142.
- The first call matches the signature taking a single `String` because that is the most specific match.
- The second call matches the signature, taking two `String` parameters since that is an exact match.
- It isn't until the third call that the varargs version is used since there are no better matches.

```
public class Glider {
    public static String glide(String s) {
        return "1"; }

    public static String glide(String... s) {
        return "2"; }

    public static String glide(Object o) {
        return "3"; }

    public static String glide(String s, String t) {
        return "4"; }

    public static void main(String[] args) {
        System.out.print(glide("a"));
        System.out.print(glide("a", "b"));
        System.out.print(glide("a", "b", "c"));
```

2) OVERLOADING METHODS

- As accommodating as Java is with trying to find a match, it will only do one conversion:

```
public class TooManyConversions {
    public static void play(Long l) {}

    public static void play(Long... l) {}

    public static void main(String[] args) {
        play(4); // DOES NOT COMPILE
        play(4L); // calls the Long version
    }
```

- Here we have a problem. Java is happy to convert the `int` 4 to a `long` 4 or an `Integer` 4.
- **It cannot handle converting in two steps to a long and then to a Long**.
- If we had `public static void play(Object o) { }`, it would match because only one conversion would be necessary: from `int` to `Integer`.
- An Integer is an Object, as we'll see a bit later.