8

Creating and Using Methods

ORACLE

Objectives

After completing this lesson, you should be able to:

- Add an argument to a method
- Instantiate a class and call a method
- Overload a method
- Work with static methods and variables
- Convert data values using `Integer`, `Double`, and `Boolean` object types

Topics

- Using methods and constructors
- Method arguments and return values
- Using static methods and variables
- Understanding how arguments are passed to a method
- Overloading a method

Basic Form of a Method

The void keyword indicates that the method does not return a value.

Empty parentheses indicate that no arguments are passed to the method.

```
1 public void display ()  {
2       System.out.println("Shirt description:" + description);
3       System.out.println("Color Code: " + colorCode);
4       System.out.println("Shirt price: " + price);
5 } // end of display method
```

This is an example of a simple method that does not receive any arguments or return a value.

Calling a Method from a Different Class

```
1  public class ShoppingCart {
2     public static void main (String[] args) {
3          Shirt myShirt = new Shirt();
4          myShirt.display();
5     }
6  }
```

Reference variable

Dot operator

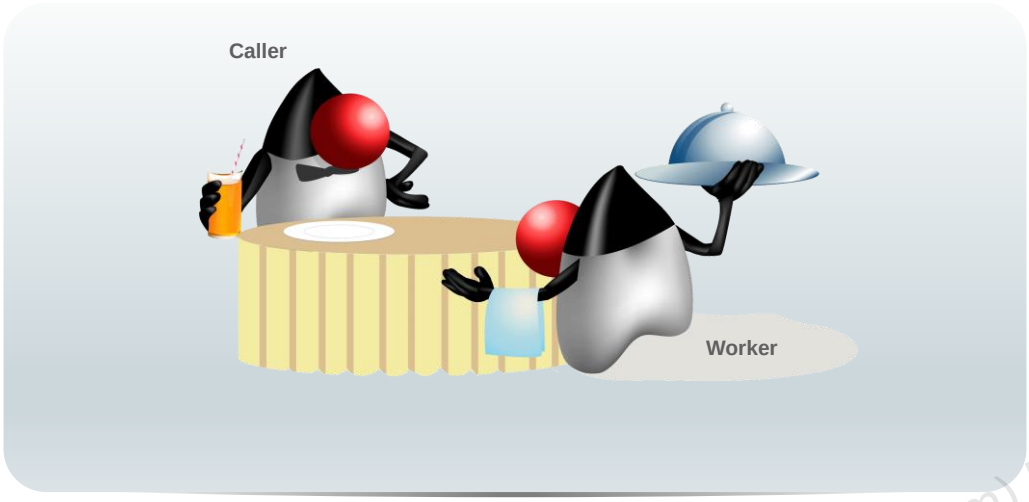
Method

Output:

```
Item description:-description required-
Color Code: U
Item price: 0.0
```

In the example in this slide, `display` is called by typing the reference variable for the object, the dot operator, followed by the method to be called. The default values, as set in the `Shirt` constructor, are displayed.

Caller and Worker Methods



In the previous example, the `ShoppingCart` class calls the `display` method on a `Shirt` object from within the `main` method. The `main` method is referred to as the *calling method* because it is invoking or "calling" another method to do some work. Conversely, the `display` method is referred to as the *worker method* because it does some work for the `main` method.

When a calling method calls a worker method, the calling method stops execution until the worker method is done. After the worker method has completed, program flow returns to the point after the method invocation in the calling method.

A Constructor Method

A constructor method is a special method that is invoked when you create an object instance.

- It is called by using the `new` keyword.
- Its purpose is to instantiate an object of the class and store the reference in the reference variable.

```
Shirt myShirt = new Shirt();
```

Constructor method is called.

- It has a unique method signature.

<modifier> ClassName()

A constructor is invoked using the `new` keyword. Its job is to instantiate an object of the class and to provide a reference to the new object. If you do not write your own constructor in a class, Java will provide one for you. The constructor's name is the same as the class name.

In the `Shirt` example above, the reference returned by the `Shirt` constructor is assigned to the `myShirt` reference variable.

Writing and Calling a Constructor

```
1  public static void main(String[] args){
2    Shirt myShirt = new Shirt()
3  }
```

```
1  public class Shirt {
2    //Fields
3    public String description;
4    public char colorCode;
5    public double price;
6
7    //Constructor
8    public Shirt(){
9        description = "--description required--";
10       colorCode = 'U'
11       price = 0.00;
12   }
13
14   //Methods
15   public void display(){
16      System.out.println("Shirt description:" + description);
17      System.out.println("Color Code: " + colorCode);
18      System.out.println("Shirt price: " + price);
19   }…
```

The constructor is the first method called when an object is instantiated. Its purpose is primarily to set default values.

Calling a Method in the Same Class

```
1  public class Shirt {
2    public String description;
3    public char colorCode;
4    public double price;
5
6    public Shirt(){
7        description = "--description required--";
8        colorCode = 'U'
9        price = 0.00;
10
11       display();          //Called normally
12       this.display();     //Called using the 'this' keyword
13   }
14
15   public void display(){
16      System.out.println("Shirt description:" + description);
17      System.out.println("Color Code: " + colorCode);
18      System.out.println("Shirt price: " + price);
19   }
20 …
```

Calling a method in the same class is very straightforward. You can simply use the method name without a reference. This is the same as when accessing a field; you can simply use the field name.

However, if you have local variables with similar names and you want to make it obvious that your code is accessing a field or method of the current object, you can use the `this` keyword with dot notation. `this` is a reference to the current object.

In this example, the display method is called twice from the constructor.

Topics

- Using constructors and methods
- Method arguments and return values
- Using static methods and variables
- Understanding how arguments are passed to a method
- Overloading a method

Method Arguments and Parameters

- An **argument** is a value that is passed during a method call:

```
Calculator calc = new Calculator();
double denominator = 2.0

calc.calculate(3, denominator);        //should print 1.5
```

Arguments

- A **parameter** is a variable defined in the method declaration:

```
public void calculate(int x, double y){
    System.out.println(x/y);
}
```

3 2.0

Parameters

Note: A value passed into the method when it is called is called an *argument*, whereas a variable that is defined in the method declaration is called a *method parameter*.

In this example, 3 and 2.0 are passed to be the values of x and y within the `calculate` method.

Method Parameter Examples

- Methods may have any number or type of parameters:

```
public void calculate0(){
    System.out.println("No parameters");
}
```

```
public void calculate1(int x){
    System.out.println(x/2.0);
}
```

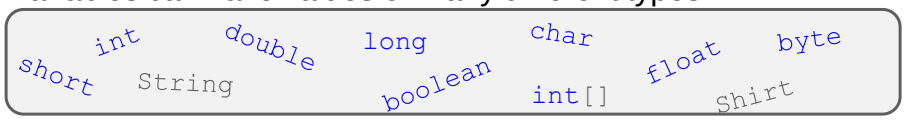
```
public void calculate2(int x, double y){
    System.out.println(x/y);
}
```

```
public void calculate3(int x, double y, int z){
    System.out.println(x/y +z);
}
```

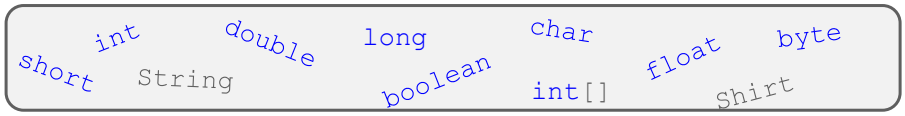
Methods can take any number of parameters and use these values within the method code block.

Method Return Types

- Variables can have values of many different types:



- Method calls can also return values of many different types:



- How to make a method return a value:
  - Declare the method to be a non-void return type.
  - Use the keyword `return` within a method, followed by a value.

Variables may have values of many different types, including primitive data types, objects, and arrays.

Likewise, methods may return values of many different types, including primitive data types, objects, and arrays.

Note: Constructors are special. They cannot have a return type, not even void.

Method Return Types Examples

- Methods must `return` data that matches their return type:

```
public void printString(){
    System.out.println("Hello");
}
```

Void methods cannot return values in Java.

```
public String returnString(){
    return("Hello");
}
```

```
public int sum(int x, int y){
    return(x + y);
}
```

```
public boolean isGreater(int x, int y){
    return(x > y);
}
```

Void methods and constructors should not have a `return` statement. Void methods are incapable of returning a value in Java. The type of value a method returns must match the return type you declare. For instance, a `boolean` type method must return a `boolean`. A `String` type method must return a `String`.

Method Return Animation

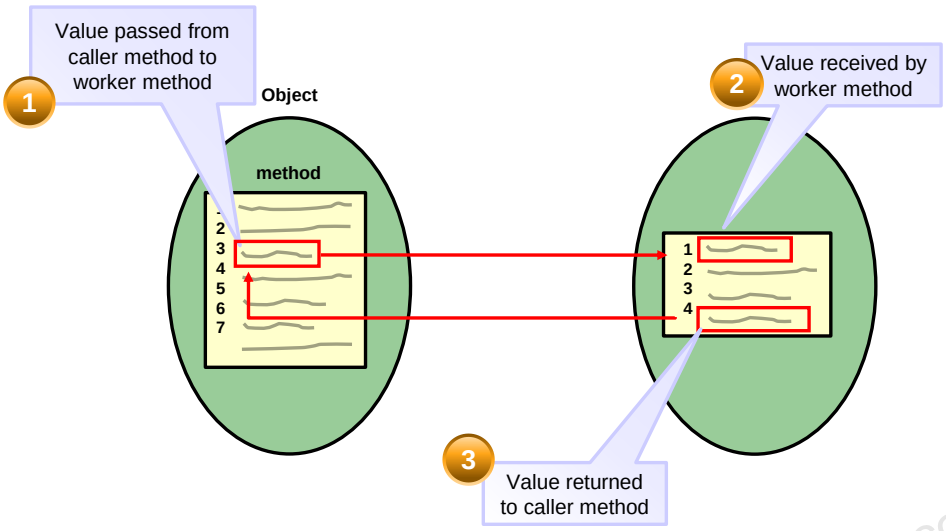
- The following code examples produce equivalent results:

```
public static void main(String[] args){
    int num1 = 1, num2 = 2;
    int result = num1 + num2;
    System.out.println(result);
}
```

```
public static void main(String[] args){
    int num1 = 1, num2 = 2;
    int result = sum(num1, num2);
    System.out.println(result);
}
public int sum(int x, int y){
    return(x + y);
}
```

In the top example, `num1` and `num2` are added together. In the bottom example, this logic is put into the `sum` method. Values are passed to the `sum` method and added, with the resulting integer value being passed back and assigned to the `result` variable.

Passing Arguments and Returning Values



More Examples

```
1 public void setCustomerServices() {
2    String message ="Would you like to hear about "
3          +"special deals in your area?";
4    if (cust.isNewCustomer()) {
5
6        cust.sendEmail(message);
7    }
8 }
```

```
1 public class Customer{
2     public boolean isNew;
3
4     public boolean isNewCustomer(){
5         return isNew;
6     }
7     public void sendEmail(String message){
8         // send email
9     }
10 }
```

Return a boolean

String argument required

Here you see a caller method, `setCustomerServices`, invoking worker methods in the `Customer` class.

- The example at the bottom of the slide shows the `Customer` class, which defines two methods:
  - `isNewCustomer` is defined with a return value of type `boolean`, but it does not define any input parameters.
  - `sendEmail` is defined with an input parameter of type `String`, called message. This method does not return a value.
- The example at top of the slide shows the `setCustomerServices` method in the `ShoppingCart` class invoking the methods of a Customer object by using dot notation (`object_reference.method`).
  - In line 4, `isNewCustomer` is called on the `cust` object reference. Because the method returns a `boolean`, the method invocation becomes a `boolean` expression evaluated by the `if` statement.
  - In line 6, `sendEmail` is called on the `cust` object reference, passing the message string as an argument.

Code Without Methods

```
1  public static void main(String[] args){
2    Shirt shirt01 = new Shirt();
3    Shirt shirt02 = new Shirt();
4    Shirt shirt03 = new Shirt();
5    Shirt shirt04 = new Shirt();
6
7    shirt01.description = "Sailor";
8    shirt01.colorCode = 'B';
9    shirt01.price = 30;
10
11   shirt02.description = "Sweatshirt";
12   shirt02.colorCode = 'G';
13   shirt02.price = 25;
14
15   shirt03.description = "Skull Tee";
16   shirt03.colorCode = 'B';
17   shirt03.price = 15;
18
19   shirt04.description = "Tropical";
20   shirt04.colorCode = 'R';
21   shirt04.price = 20;
22 }
```

Why are methods useful? To answer that question, take a look at this code without methods. For every instance of the `Shirt` object that you want to create, you need many more lines of code to edit each object. Methods can help this code be more efficient and less cumbersome to work with.

Better Code with Methods

```
1  public static void main(String[] args){
2    Shirt shirt01 = new Shirt();
3    Shirt shirt02 = new Shirt();
4    Shirt shirt03 = new Shirt();
5    Shirt shirt04 = new Shirt();
6
7    shirt01.setFields("Sailor", 'B', 30);
8    shirt02.setFields("Sweatshirt", 'G', 25);
9    shirt03.setFields("Skull Tee", 'B', 15);
10   shirt04.setFields("Tropical", 'R', 20);
11 }
```

```
1  public class Shirt {
2    public String description;
3    public char colorCode;
4    public double price;
5
6    public void setFields(String desc, char color, double price){
7        this.description = desc;
8        this.colorCode = color;
9        this.price = price;
10   }
11 …
```


With a little bit of extra coding in the `Shirt` class, we can create a method that sets all the appropriate fields. This reduces the amount of code needed in the `main` method to create and edit `Shirt` objects.

Even Better Code with Methods

```
1  public static void main(String[] args){
2    Shirt shirt01 = new Shirt("Sailor", "Blue", 30);
3    Shirt shirt02 = new Shirt("SweatShirt", "Green", 25);
4    Shirt shirt03 = new Shirt("Skull Tee", "Blue", 15);
5    Shirt shirt04 = new Shirt("Tropical", "Red", 20);
6  }
```

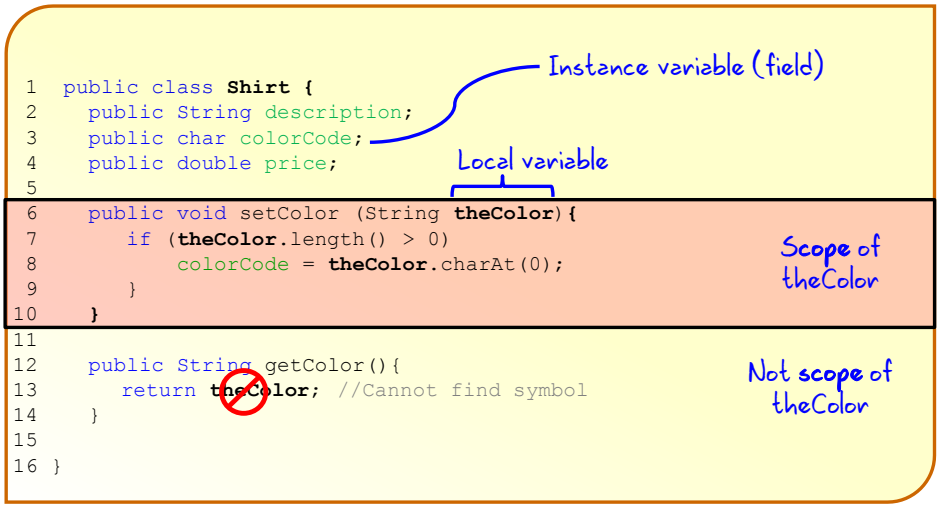
```
1  public class Shirt {
2    public String description;
3    public char colorCode;
4    public double price;
5
6    //Constructor
7    public Shirt(String desc, String color, double price){
8        setFields(desc, price);
9        setColor(color);
10   }
11   public void setColor (String theColor){
12      if (theColor.length() > 0)
13          colorCode = theColor.charAt(0);
14      }
15   }
16 }
```

Taking advantage of a `Shirt` constructor can further reduce the amount of code needed in the `main` method.

Another issue is maintenance. Imagine if you wanted to change the constructor so that the color passed in is a `String`, but the instance variable, `colorCode`, remains a `char` type. You could create a method `setColor` that receives a `String` as an argument and then modifies it so that it sets `colorCode` correctly.

Remember, methods can call other methods (as shown by the call to `setColor`).

Variable Scope



This code illustrates the scope of two different types of variables. Variables live in the block where they are defined. This is called “scope.” The scope of a variable determines its accessibility and also how long you can count on its value to persist.

- The `colorCode` variable is an instance variable, usually called a field. It is a member of the Shirt class. It is accessible from any code within this class. The value of fit is stored only during the lifespan of an instance.
- `theColor` is a local variable. It is accessible only from within the `setColor` method. The value of `theColor` is deleted from memory when the method ends. Another way of saying this is that its scope is the `setColor` method.
- Regardless of whether a local variable is declared within a method, a loop (discussed later), or an `if` statement, its scope is always the block within which it is declared.
- In the example above, the `setColor` method uses the `charAt` method of the `String` object to extract the first character in the `theColor String`. It assigns it to the `fit` instance variable, which is a char.

Note: Local variables are stored in short-term memory, called “the stack,” whereas instance variables (fields) are stored in a longer-term area of memory called “the heap.”

Advantages of Using Methods

Methods:

- Are reusable
- Make programs shorter and more readable
- Make development and maintenance quicker
- Allow separate objects to communicate and to distribute the work performed by the program

Exercise 8-1: Declare a `setColor` Method

1. Open the project `Exercise_08-1` in NetBeans.

In the `Item` class:

2. Declare a `setColor` method that takes a char as an argument (a color code) and returns a `boolean`. Return false if the `colorCode` is `' '` (a single space). Otherwise, assign the `colorCode` to the color field and return true.

In the `ShoppingCart` class:

3. Call the `setColor` method on `item1`. If it returns true, print `item1.color`. If it returns false, print an invalid color message.
4. Test the `setColor` method with both a valid color and an invalid one.

In this exercise, you declare a `setColor` method that takes a char as an argument, call the `setColor` method on `item1`, and test this method with both a valid color and invalid color.

Topics

- Using constructors and methods
- Method arguments and return values
- Using static methods and variables
- Understanding how arguments are passed to a method
- Overloading a method

Static Methods and Variables

The `static` modifier is applied to a method or variable.

It means the method/variable:

- Belongs to the *class* and is shared by all objects of that class
- Is *not unique* to an object instance
- Can be accessed without instantiating the class

Comparison:

- A **static variable** is shared by all objects in a class.
- An **instance variable** is unique to an individual object.

So far you learned how to access variables and methods by creating an object instance of the class that the variable or method belongs to. The Java language allows you to declare a variable or method as `static`. This means that you can access it *without* creating an object instance of the class. Sometimes these are referred to as *class variables* or *class methods.*

Example: Setting the Size for a New Item

```
1 public class ItemSizes {
2     static final String mSmall = "Men's Small";
3     static final String mMed = "Men's Medium";
4 }
```

Passing the static mMed variable to the setSize method

```
Item item1 = new Item();
item1.setSize(ItemSizes.mMed);
```

```
1 public class Item {
2     public String size;
3     public void setSize(String sizeArg){
4         this.size = sizeArg;
5     }
6 }
```

In the example above, the class `ItemSizes` contains two static variables of type `String`: `mSmall` and `mMed`. These are initialized to a description of a particular men's size. These values can be used without instantiating `ItemSizes`.

- The code snippet shown in the middle of the slide shows an Item object being instantiated and then the `setSize` method of the Item object is invoked, passing in `ItemSizes.mMed` as an argument.
- The code example at the bottom of the slide shows the `Item` class. It contains a `String` field, `size`. The `setSize` method requires a `String` parameter to set the `size` field.

Creating and Accessing Static Members

- To create a static variable or method:

```
static String mSmall;
static void setMSmall(String desc);
```

- To access a `static` variable or method:
  - From another class

```
ItemSizes.mSmall;
ItemSizes.setMSmall("Men's Small");
```

  - From within the class

```
mSmall;
setMSmall("Men's Small");
```

Variables and methods that are unique to an instance are referred to as *instance* variables or methods. If they are accessed from an object of another class, you qualify the reference with the object reference (`shirt01.size`).

- When accessing a static variable or method from an object of a different class, you qualify the reference with the class name as shown above: `ItemSizes.setMSmall("Men's Small")` or `ItemSizes.mSmall`
- If you are referencing the static variable or method from within the class, there is no need to qualify it.
- The `main` method is an example of a static method. As you know, it is used as the entry point to an application. Because the main method is static, the Java runtime can implicitly invoke it on the class without first instantiating the class.

When to Use Static Methods or Fields

- Performing the operation on an individual object or associating the variable with a specific object type is not important.
- Accessing the variable or method before instantiating an object is important.
- The method or variable does not logically belong to an object, but possibly belongs to a utility class, such as the `Math` class, included in the Java API.
- Using constant values (such as `Math.PI`)

Some Rules About Static Fields and Methods

- Instance methods can access static methods or fields.
- Static methods cannot access instance methods or fields. Why?

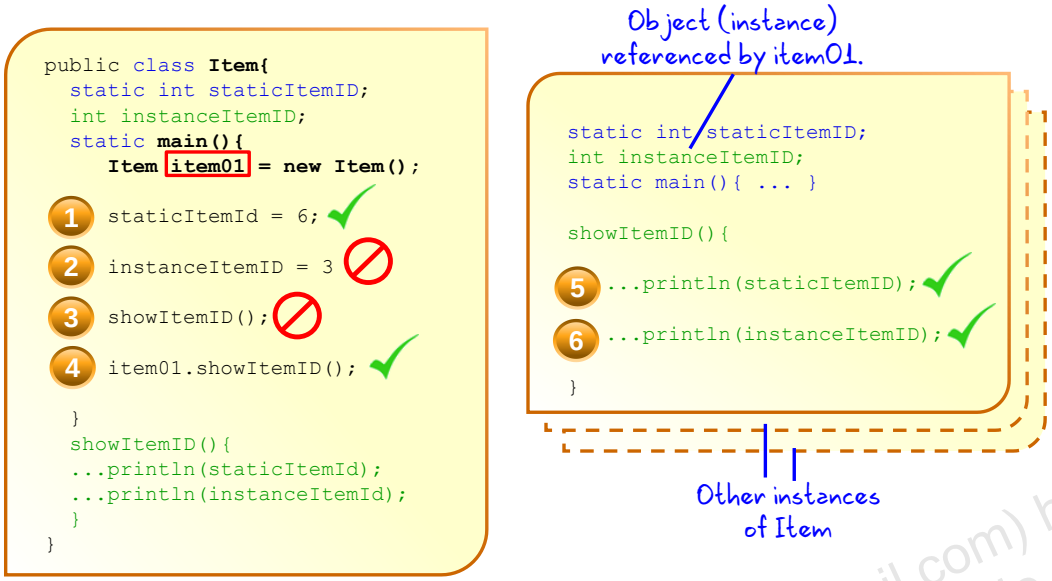
```
1 public class Item{
2   int itemID;
3   public Item(){
4      setId();
5   }
6   static int getID(){
7      // whose itemID??
8   }
```

The code example above illustrates why a static method is not allowed to access an instance method or field.

- `itemID` is an instance variable. That means that each Item object has its own (presumably) unique `itemID`. In this example, its value is set in the constructor.
- The `getID` method is static, so it can be invoked even if there are no `Item` objects created.

Instance methods and fields are only available by referencing the individual object instance. \

Static Fields and Methods Versus Instance Fields and Methods



The code example above shows a more complex example of an `Item` class that has an instance variable `instanceItemID` and a static variable `staticItemID`. In its `main` method, it instantiates an object referenced by `item01`. Look at the six lines of code and see the explanations below for why some work and some do not.

1. `staticItemID` is a static variable, and referenced from within a static method, `main`, so it does not need to access an instance.
2. `instanceItemID` is an instance variable, and referenced from within a static method, `main`, so it cannot be accessed unless a reference points to the particular object whose instance variable needs to be set.
3. `showItemID()` is a call to an instance method, and referenced from within a static method, `main`, so it cannot be accessed without a reference.
4. `item01.showItemID()` is a call to an instance method, but in this case the reference points to the particular object whose instance method needs to be called.
5. `...println(staticItemID)` refers to a static variable, but it is referred to from an instance. Instances can always access static variables.
6. `...println(instanceItemID` refers to an instance variable, but it is referred to from an instance. No object reference is given, so it accesses the instance variable on the object itself.

Static Methods and Variables in the Java API

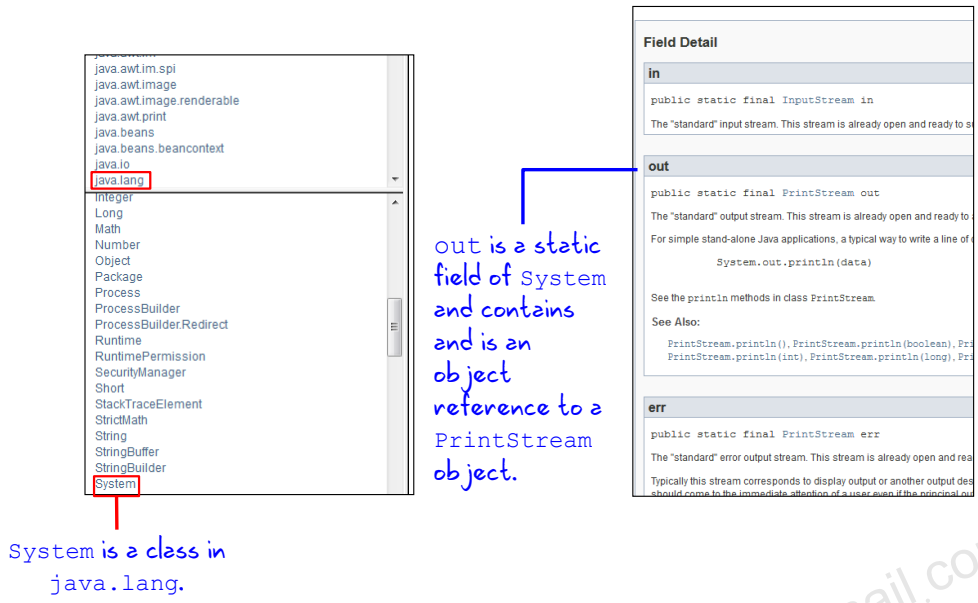
Examples:

- Some functionality of the `Math` class:
  - Exponential
  - Logarithmic
  - Trigonometric
  - Random
  - Access to common mathematical constants, such as the value PI (`Math.PI`)
- Some functionality of the `System` class:
  - Retrieving environment variables
  - Access to the standard input and output streams
  - Exiting the current program (`System.exit` method)

Certain Java class libraries, such as the `System` and the `Math` class, contain only static methods and variables. The `System` class contains utility methods for handling operating system–specific tasks. (They do not operate on an object instance.) For example, the `getProperties()` method of the `System` class gets information about the computer that you are using.

The `Math` class contains utility methods for math operations. Because these methods and variables are static, you do not need to create a new object every time you want your program to do some math.

Examining Static Variables in the JDK Libraries



The next few slides show how you might use the Java API documentation to find out more about `System.out.println()`. As you will see, this is a little unusual, because the class that has the methods that you need to investigate is not `System`. Rather, it is the class that is the type of the `out` field of the `System` object. Consider the following:

`System` is a class (in `java.lang`).

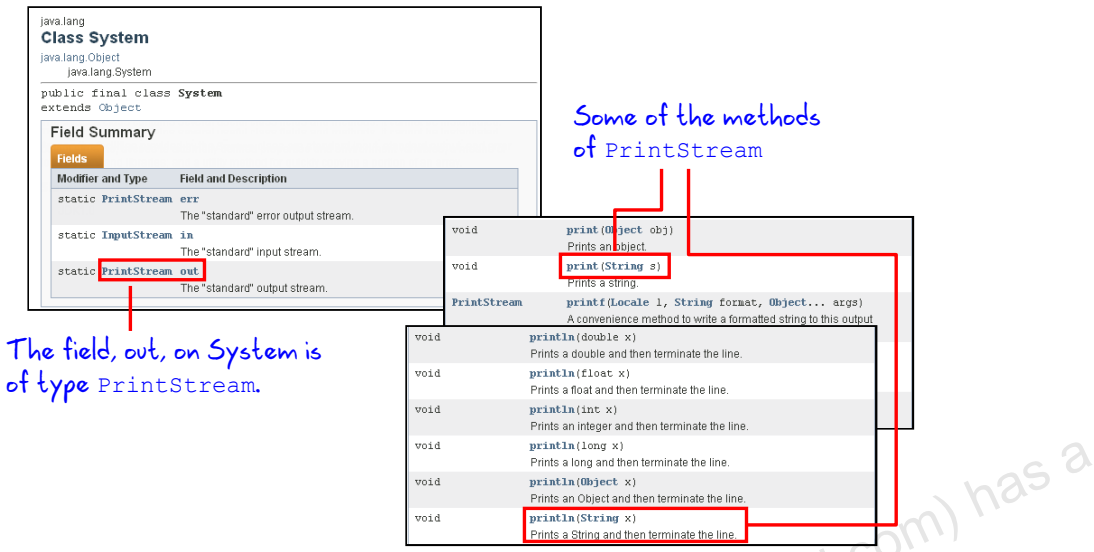
`out` is a static field of `System`. This is the reason that you reference it from the class name, not from an object instance: `System.out`

`out` is a reference type that allows calling `println()` on the object type it references.

To find the documentation:

1. Go to `System` class and find the type of the `out` field.
2. Go to the documentation for that field.
3. Review the methods available.

Using Static Variables and Methods: `System.out.println`



The diagram shows the Field Summary for the class `System`. Here, you can see that there is indeed a field called `out`, and it is of type `PrintStream`. By clicking `PrintStream`, you can now see the details for that class and, if you scroll down to the Method Summary, you will find (among many other methods) the `print` method and the `println` method. The `print` method is very similar to `println`, except that it does not create a new line after printing, like `println` does.

Example:

`println("Hello"); println("Hello")` yields the following output:

```
Hello
Hello
```

`print("Hello"); print("Hello");` yields the following output:

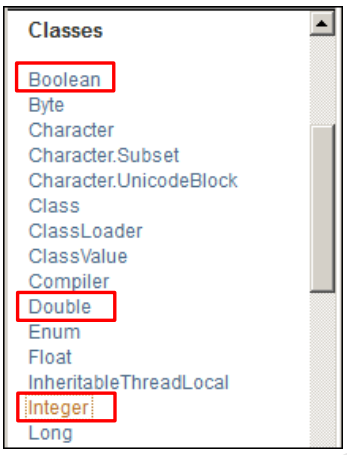
```
HelloHello
```

More Static Fields and Methods in the Java API

Java provides wrapper classes for each of the primitive data types.

- `Boolean`: Contains a single field of type `boolean`
- `Double`: Contains a single field of type `double`
- `Integer`: Contains a single field of type `int`

They also provide utility methods to work with the data.



A wrapper class is a class with the same name as one of the primitive data types. Wrapper classes are instantiated to contain a single value of the primitive type.

```
Integer myInt = Integer.valueOf(10);
```

These are very useful classes because they provide methods to help you work with the primitive values stored within.

Converting Data Values

- Methods often need to convert an argument to a different type.
- Most of the object classes in the JDK provide various conversion methods.

Examples:

- Converting a String to an `int`

```
int myInt1 = Integer.parseInt(s_Num);
```

- Converting a String to a `double`

```
double myDbl = Double.parseDouble(s_Num);
```

- Converting a String to `boolean`

```
boolean myBool = Boolean.valueOf(s_Bool);
```

The examples show static conversion methods for `Integer`, `Double`, and `Boolean`.

There are also some conversion methods for the object classes (`Integer`, `Double`, and so on) that are not static. These methods are invoked on an object reference for one of these classes and convert the value of that specific object.

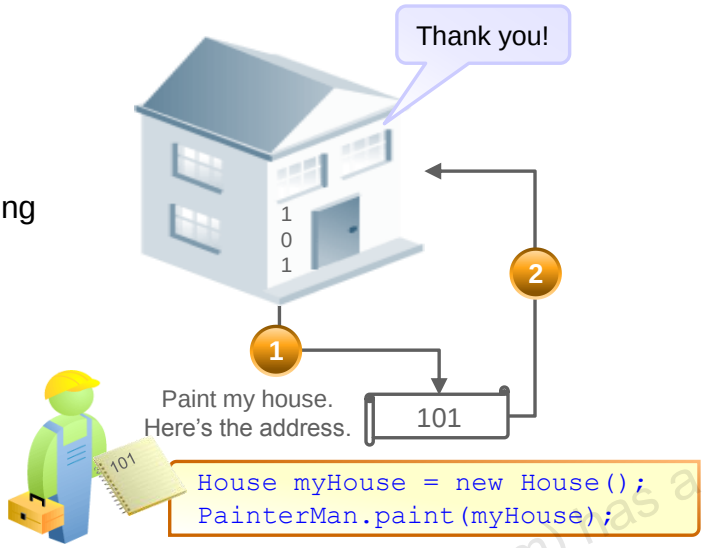
Topics

- Using constructors and methods
- Method arguments and return values
- Using static methods and variables
- Understanding how arguments are passed to a method
- Overloading a method

Passing an Object Reference

An object reference is similar to a house address. When it is passed to a method:

- The object itself is not passed
- The method can access the object using the reference
- The method can act upon the object



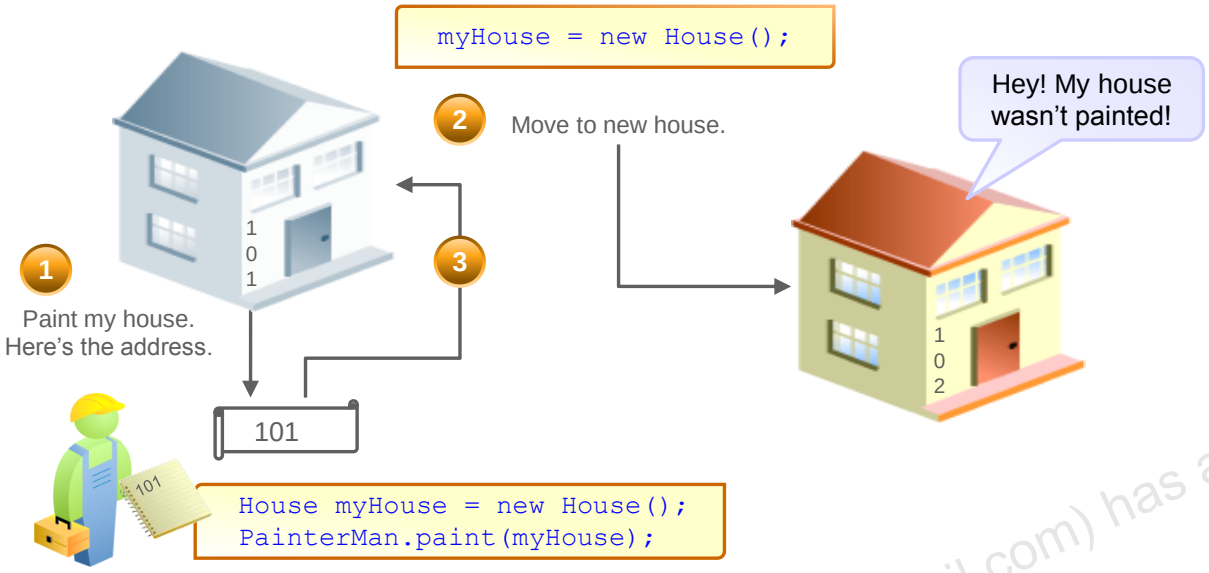
An object reference is not the same as the object. It simply provides a reference for access to that object. This is similar to the way a house address provides directions for finding a particular house.

In the graphic above, the house (call it `myHouse`) has an address (the `myHouse` reference) of `101`. When the painter gets this address, he jots it down in his notebook (he makes a copy of it). This enables the house painter to find the house and paint it.

When you send an object reference as an argument to a method, you are sending a *copy* of the reference—not the object nor the actual reference.

The receiving method has the information it needs to act directly upon the object itself.

What If There Is a New Object?



Suppose that the owner of the house moves to another house before the job is finished. Will the painter be able to find the owner's new house in order to paint it? The object reference (`myHouse`) has changed to point to a new house, but the notation in the painter's notebook still refers to the old house. If the owner expects the new house to be painted, he or she will be disappointed.

A Shopping Cart Code Example

```
1 public class ShoppingCart {
2     public static void main (String[] args) {
3         Shirt myShirt = new Shirt();
4         System.out.println("Shirt color: " + myShirt.colorCode);
5         changeShirtColor(myShirt, 'B');
6         System.out.println("Shirt color: " + myShirt.colorCode);
7     }
8     public static void changeShirtColor(Shirt theShirt, char color) {
9         theShirt.colorCode = color;     }
10 }
```

theShirt is a new reference of type Shirt.

Output:

```
Shirt color: U
Shirt color: B
```

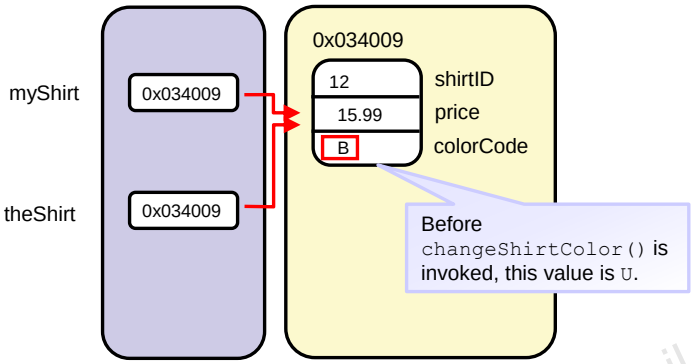

When a method is invoked, the values of the arguments are used to initialize the parameter variables before the body of the method is executed. This is true for both primitive types and reference types. (Objects are not passed to methods.)

In the example shown in the slide, the reference `myShirt` is passed by value into the `changeShirtColor` method. The reference, `theShirt` is assigned the value of the `myShirt` reference (the address). They now both point to the same object, so the change to the color made using `theShirt` is printed out by accessing `myShirt.color`.

Note: The call to the `changeShirtColor` method is made from the `main` method, which is static. Remember that a static method can only access other static methods. The `changeShirtColor` method is also static.

Passing by Value

```
Shirt myShirt = new Shirt();
changeShirtColor(myShirt, 'B');
```



The diagram in the slide shows how the value of the `myShirt` reference passed into the `changeShirtColor()` method is used to initialize a new `Shirt` reference (in this case, called `theShirt`). Remember that when a new `Shirt` is created, the `colorCode` is initialized to “U”.

Reassigning the Reference

```
1 public class ShoppingCart {
2     public static void main (String[] args) {
3         Shirt myShirt = new Shirt();
4         System.out.println("Shirt color: " + myShirt.colorCode);
5         changeShirtColor(myShirt, 'B');
6          System.out.println("Shirt color: " + myShirt.colorCode);
7     }
9     public static void changeShirtColor(Shirt theShirt, char color) {
10        theShirt = new Shirt();
11        theShirt.colorCode = color;
12 }
```

Output:

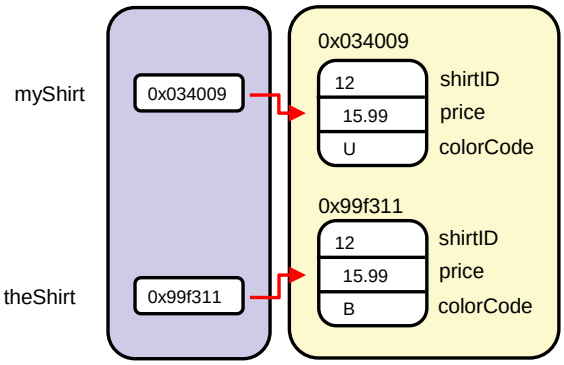
```
Shirt color: U
Shirt color: U
```

Here is another example with a small change in the code of the `changeShirtColor()` method. In this example, the reference value passed into the method is assigned to a *new* shirt. The reference now points to a different `Shirt` object than the `myShirt` reference does. As before, the `Shirt.color` is changed to '`B`'. The `println` method called on line 6 shows the color of the `myShirt` object still is '`U`' (Unset). These references point to two different Shirt objects.

This illustrates that the reference `myShirt` is indeed passed by value. Changes made to a reference passed into a worker method (reassignment to a different object, for instance) do not affect the references in the calling method.

Passing by Value

```
Shirt myShirt = new Shirt();
changeShirtColor(myShirt, 'B');
```



The diagram in the slide shows the situation that results from the code in the previous slide.

When `myShirt` is passed into the `changeShirtColor()` method, a new reference variable, `theShirt`, is initialized with the value of `myShirt`. Initially, this reference points to the object that the `myShirt` reference points to. But after a new `Shirt` is assigned to `theShirt`, any changes made using `theShirt` affect only this new `Shirt` object.

Topics

- Using constructors and methods
- Method arguments and return values
- Using static methods and variables
- Understanding how arguments are passed to a method
- Overloading a method

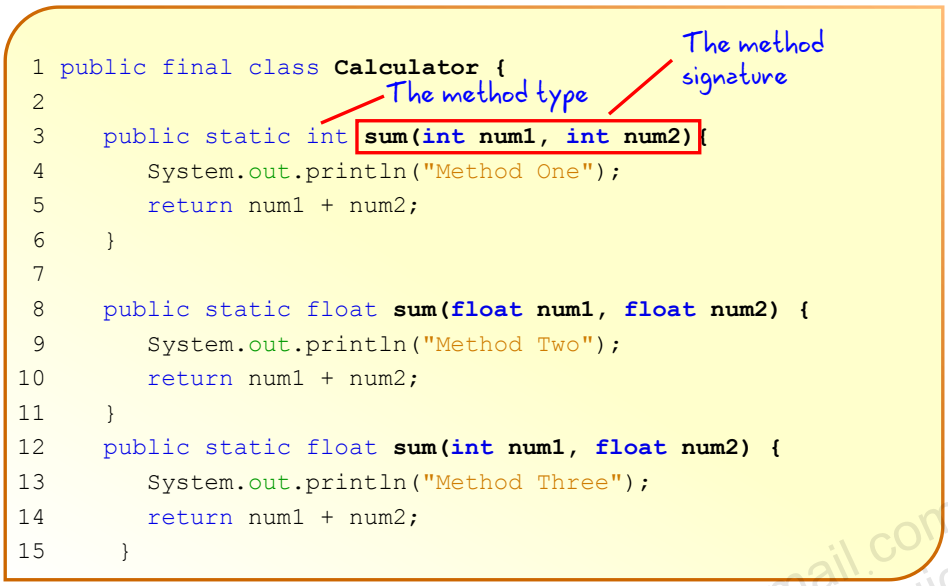
Method Overloading

Overloaded methods:

- Have the same name
- Have different signatures
  - The **number** of parameters
  - The **types** of parameters
  - The **order** of parameters
- May have different functionality or similar functionality
- Are widely used in the foundation classes

In the Java programming language, a class can contain several methods that have the same name but different arguments (so the method signature is different). This concept is called *method overloading*. Just as you can distinguish between two students named “Jim” in the same class by calling them “Jim in the green shirt” and “Jim with the beeper,” you can distinguish between two methods by their name and arguments.

Using Method Overloading



The example in the slide shows three methods to add two numbers, such as two `int` types or two `float` types. With method overloading, you can create several methods with the same name and different signatures.

The first `sum` method accepts two `int` arguments and returns an `int` value. The second `sum` method accepts two `float` arguments and returns a `float` value. The third `sum` method accepts an `int` and a `float` as arguments and returns a `float`.

The callout shows the part of the method declaration that is called the *method signature*.

The method signature of a method is the unique combination of the method name and the number, types, and order of its parameters. The method signature does not include the return type. To invoke any of the `sum` methods, the compiler compares the method signature in your method invocation against the method signatures in a class.

Using Method Overloading

```
1  public class CalculatorTest {
2
3      public static void main(String[] args) {
4
5          int totalOne = Calculator.sum(2, 3);
6          System.out.println("The total is " + totalOne);
7
8          float totalTwo = Calculator.sum(15.99F, 12.85F);
9          System.out.println(totalTwo);
10
11         float totalThree = Calculator.sum(2, 12.85F);
12         System.out.println(totalThree);
13     }
14 }
```

The code example in the slide has a `main` method that invokes each of the previous `sum` methods of the `Calculator` class.

Method Overloading and the Java API

Method	Use
`void println()`	Terminates the current line by writing the line separator string
`void println(boolean x)`	Prints a boolean value and then terminates the line
`void println(char x)`	Prints a character and then terminates the line
`void println(char[] x)`	Prints an array of characters and then terminates the line

Many methods in the Java API are overloaded, including the `System.out.println` method. The table in the slide shows four variations of the `println` method.

Exercise 8-2: Overload a `setItemFields` Method, Part 1

1. Open the project Practice_08-2 in NetBeans.

In the `Item` class:

2. Write a `setItemFields` method that takes three arguments and assigns them to the `desc`, `quantity`, and `price` fields. The method returns void.
3. Create an overloaded `setItemFields` method to take four arguments and return an int. The method assigns all four fields. A `' '` (a single space) is an invalid value for a colorCode argument.
   - If the `colorCode` argument is invalid, return -1 without assigning the value.
   - If the `colorCode` is valid, assign the `colorCode` field and then assign the remaining fields by calling the three-argument method.

.

In this exercise, you create an overloaded method in the Item class. Then, you invoke these from the ShoppingCart class.

Exercise 8-2: Overload a `setItemFields` Method, Part 2

In the `ShoppingCart` class:

4. Call the 3-argument `setItemFields` method and then call `item1.displayItem()`.
5. Call the 4-argument `setItemFields` method. Check the return value.
   - If the return value < 0, print an invalid color code message.
   - Otherwise, call `displayItem()`.

In this exercise, you create an overloaded method in the Item class. Then, you invoke these from the ShoppingCart class.

Quiz

Which method corresponds to the following method call?

```
myPerson.printValues(100, 147.7F, "lavender");
```

a. `public void printValues (int i, float f)`

b. `public void printValues (i, float f, s)`

c. `public void printValues (int i, float f, String s)`

d. `public void printValues (float f, String s, int i)`

Answer: c

Summary

In this lesson, you should have learned how to:

- Add an argument to a method
- Instantiate a class and call a method
- Overload a method
- Work with static methods and variables
- Convert data values using `Integer`, `Double`, and `Boolean` object types

Practices Overview

- 8-1: Using Methods
- 8-2: Creating Game Data Randomly
- 8-3: Creating Overloaded Methods