Altar Code Labs

Java an Object Oriented Language

Java SE 2/4

UP ACADEMY
LEARNING FOR SUCCESS

CONTENTS

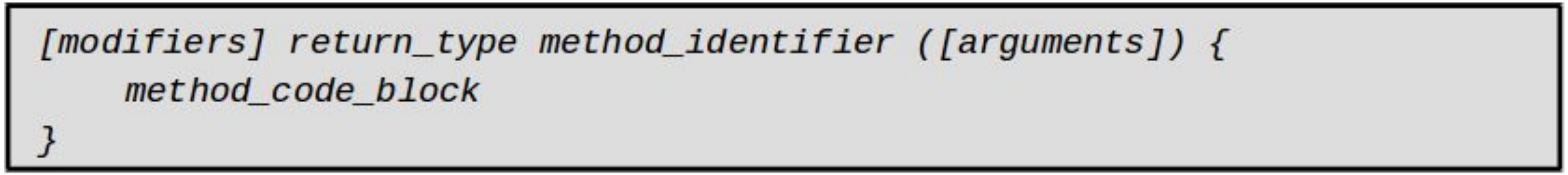
1. Methods
    - Methods
    - Static keyword
    - Method overloading
2. Object Oriented
    - Inheritance
    - Abstract classes
    - Polymorphism
    - Interfaces

Methods

- *Methods*
- *Static keyword*
- *Method overloading*

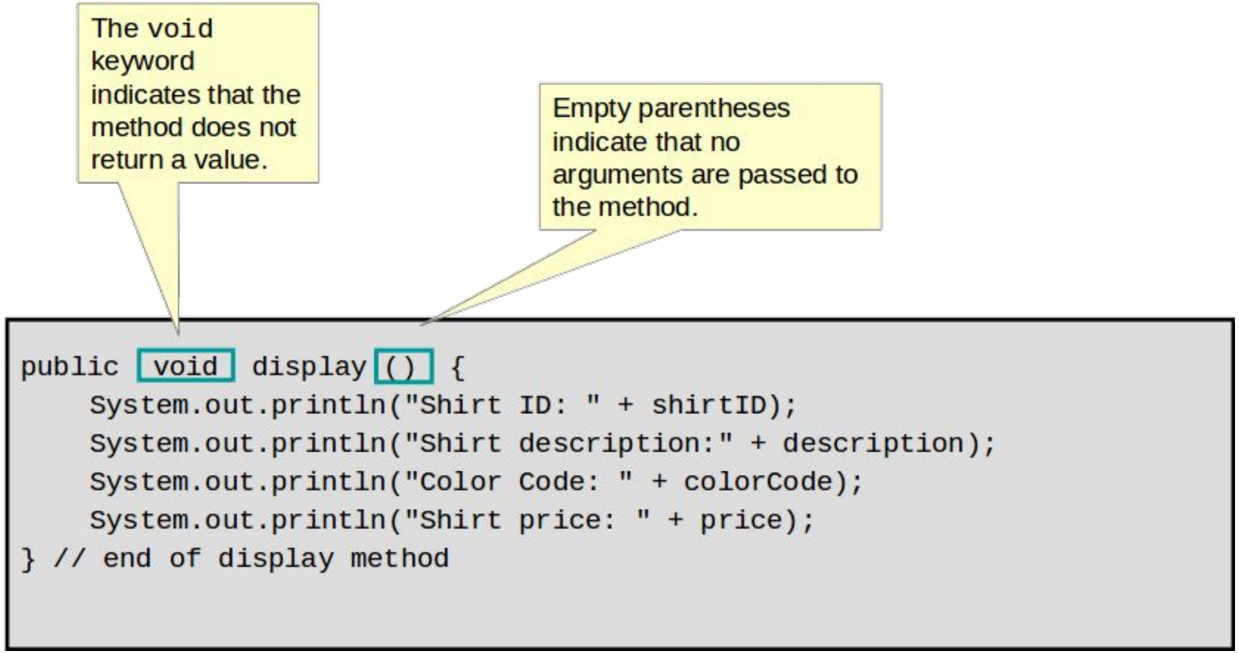
Creating Methods

Syntax:



```
[modifiers] return_type method_identifier ([arguments]) {
    method_code_block
}
```

Basic Form of a Method

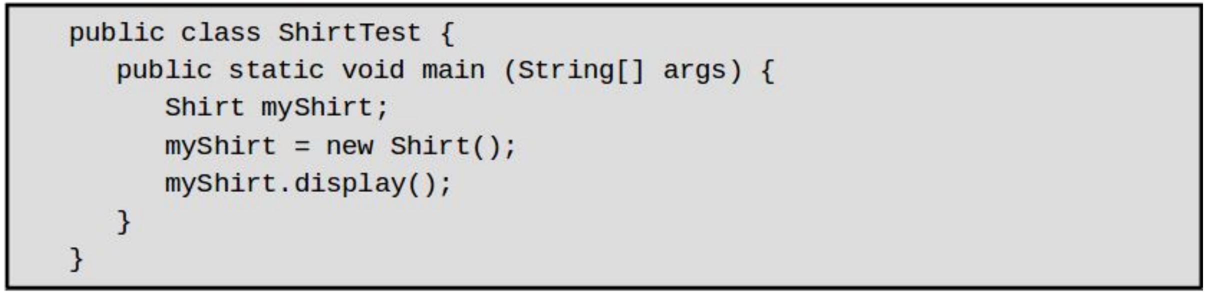


The void keyword indicates that the method does not return a value.

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

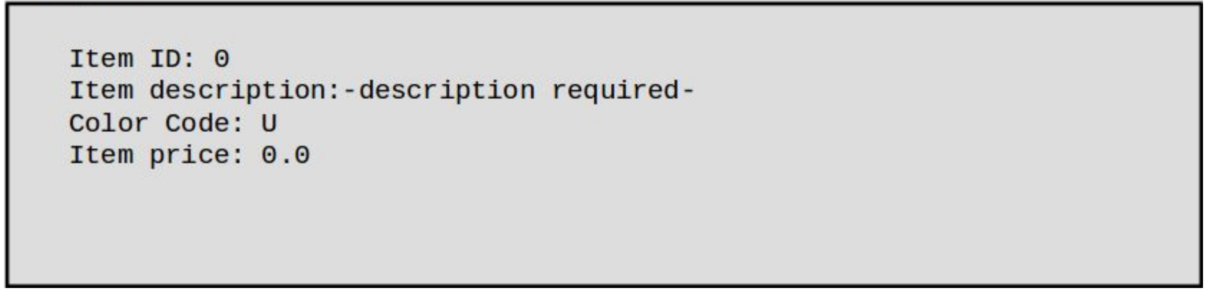
```
public  void  display ()  {
    System.out.println("Shirt ID: " + shirtID);
    System.out.println("Shirt description:" + description);
    System.out.println("Color Code: " + colorCode);
    System.out.println("Shirt price: " + price);
} // end of display method
```

Invoking a Method in a Different Class



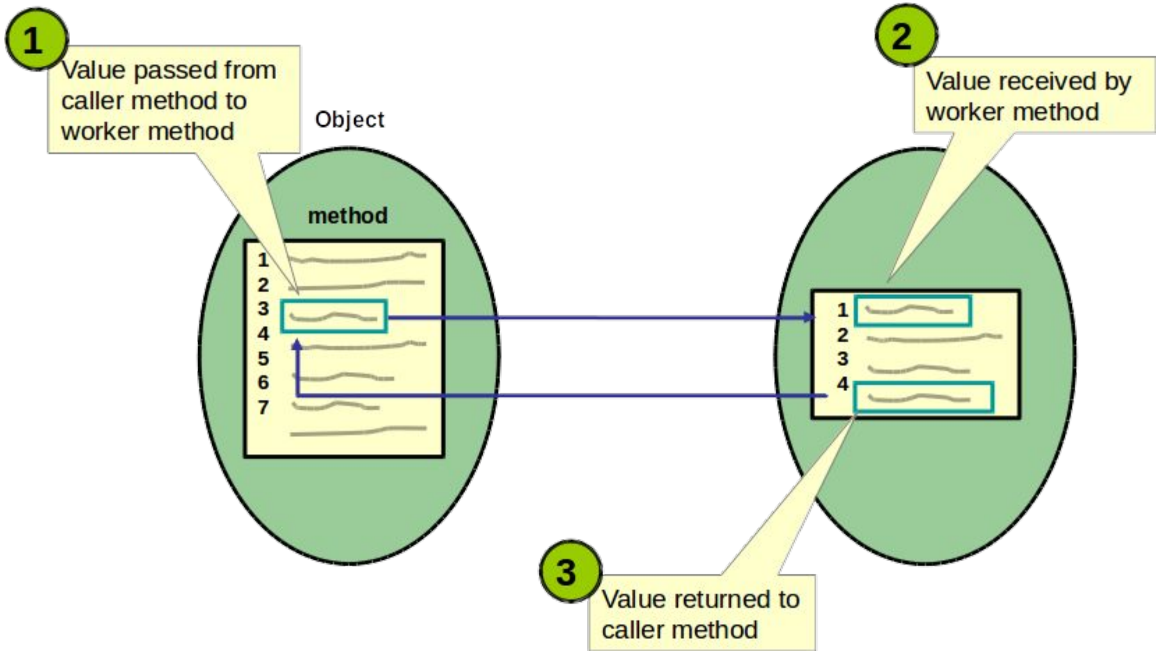
```
public class ShirtTest {
   public static void main (String[] args) {
      Shirt myShirt;
      myShirt = new Shirt();
      myShirt.display();
   }
}
```

Output:

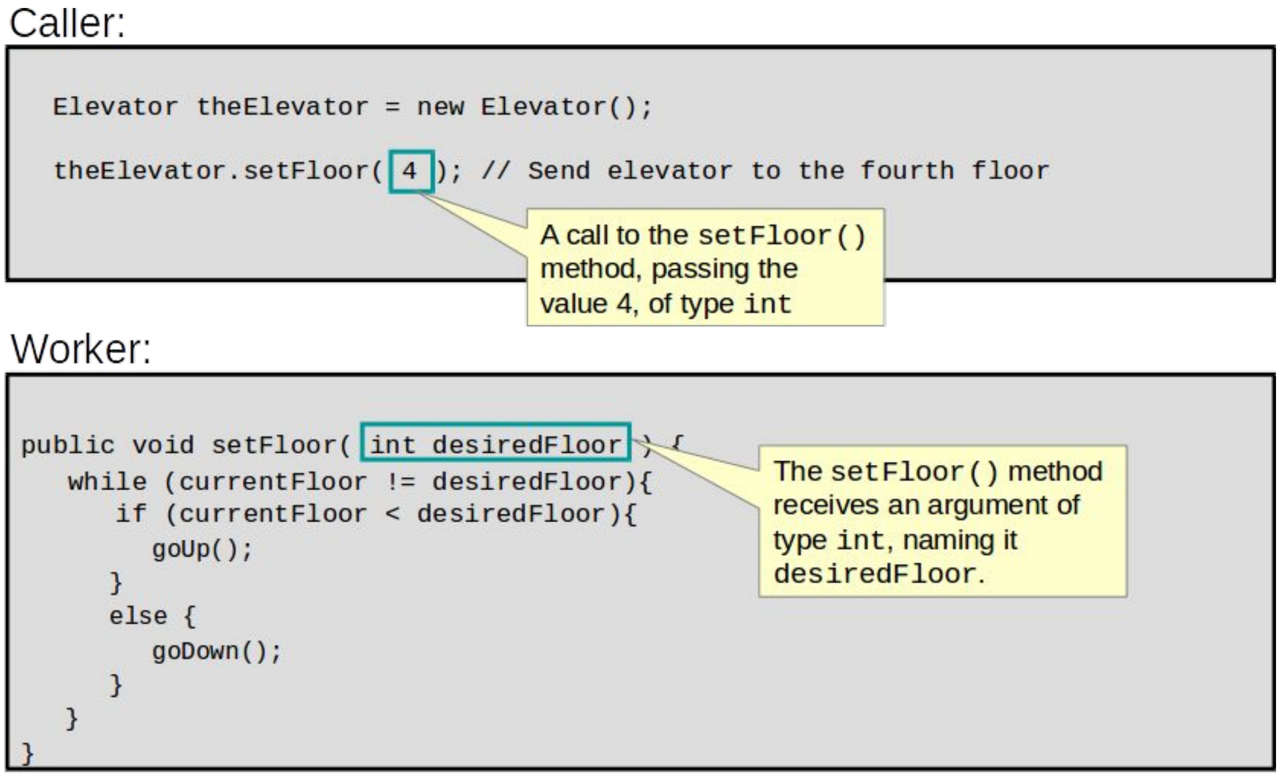


```
Item ID: 0
Item description:-description required-
Color Code: U
Item price: 0.0
```

Passing Arguments and Returning Values


1
Value passed from caller method to worker method
Object
method
1
2
3
4
5
6
7
2
Value received by worker method
1
2
3
4
3
Value returned to caller method

Creating a Method with a Parameter



Caller:

```
Elevator theElevator = new Elevator();

theElevator.setFloor( 4 ); // Send elevator to the fourth floor
```

A call to the `setFloor()` method, passing the value 4, of type `int`

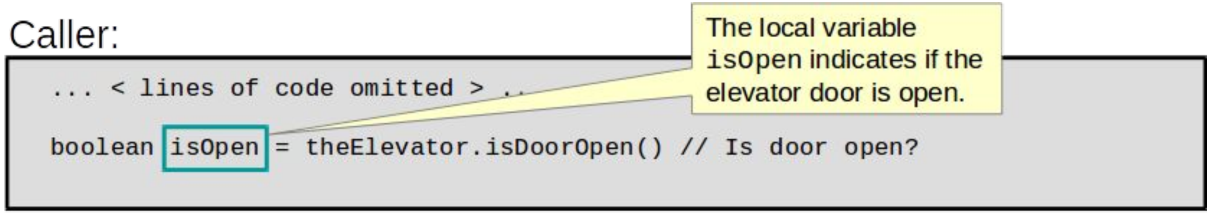
Worker:

```
public void setFloor( int desiredFloor ) {
    while (currentFloor != desiredFloor){
        if (currentFloor < desiredFloor){
           goUp();
        }
       else {
           goDown();
        }
    }
}
```

The `setFloor()` method receives an argument of type `int`, naming it `desiredFloor`.

Creating a Method with a Return Value

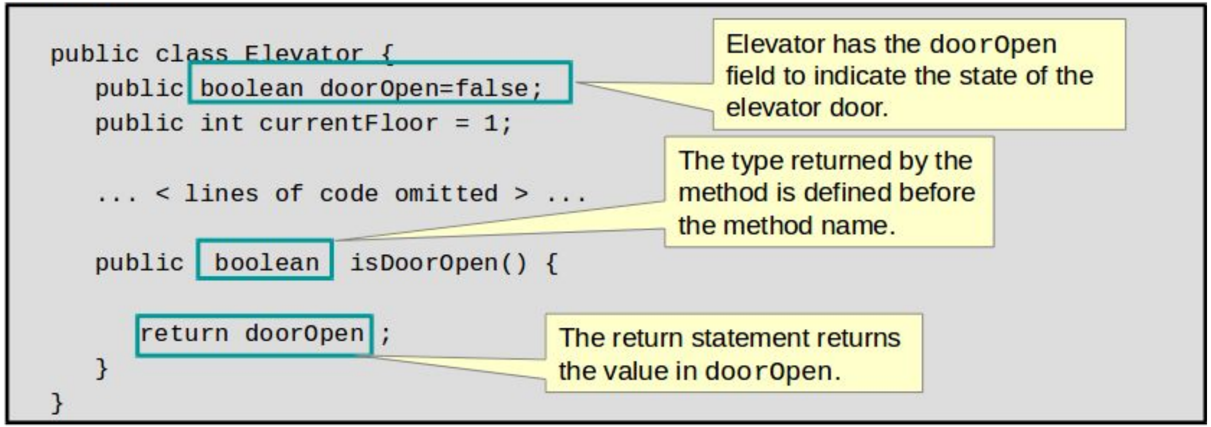
Caller:



```
... < lines of code omitted > ...

boolean isOpen = theElevator.isDoorOpen() // Is door open?
```

The local variable isOpen indicates if the elevator door is open.

Worker:



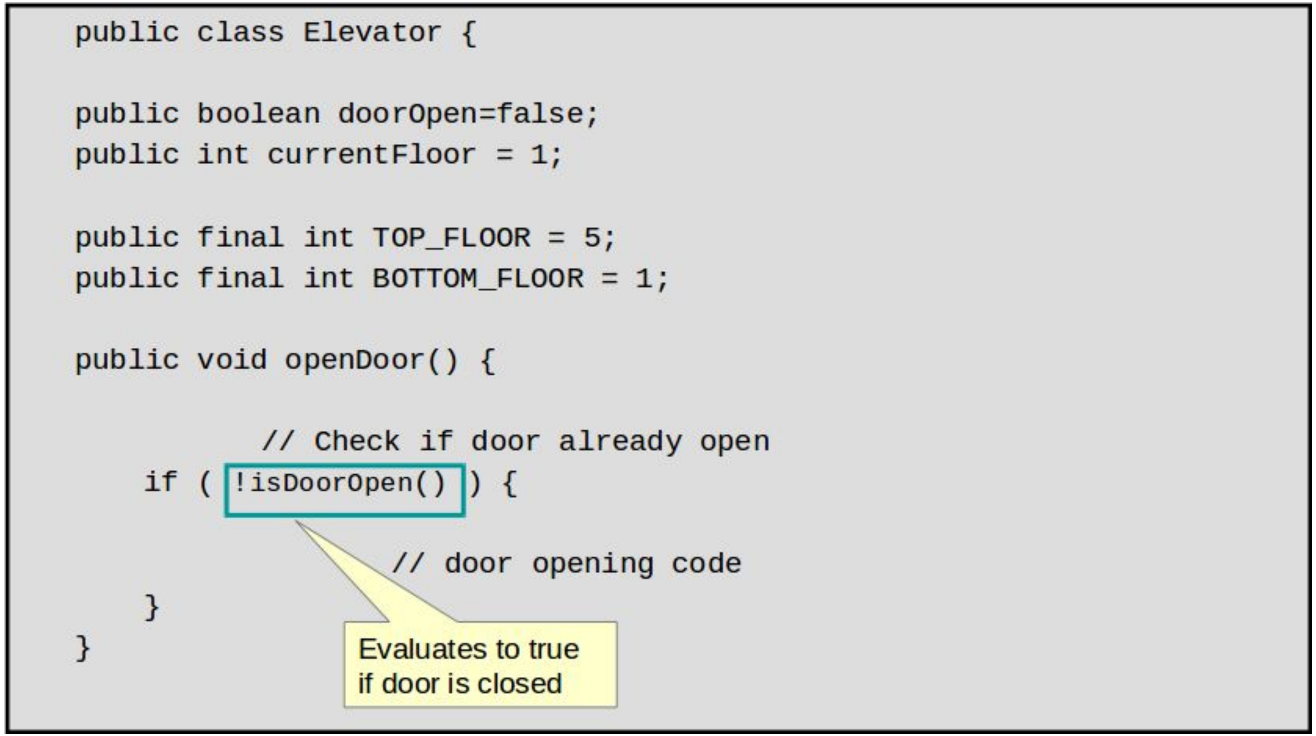
```
public class Elevator {
   public boolean doorOpen=false;
   public int currentFloor = 1;

   ... < lines of code omitted > ...

   public  boolean  isDoorOpen() {

     return doorOpen ;
   }
}
```

Elevator has the doorOpen field to indicate the state of the elevator door.

The type returned by the method is defined before the method name.

The return statement returns the value in doorOpen.

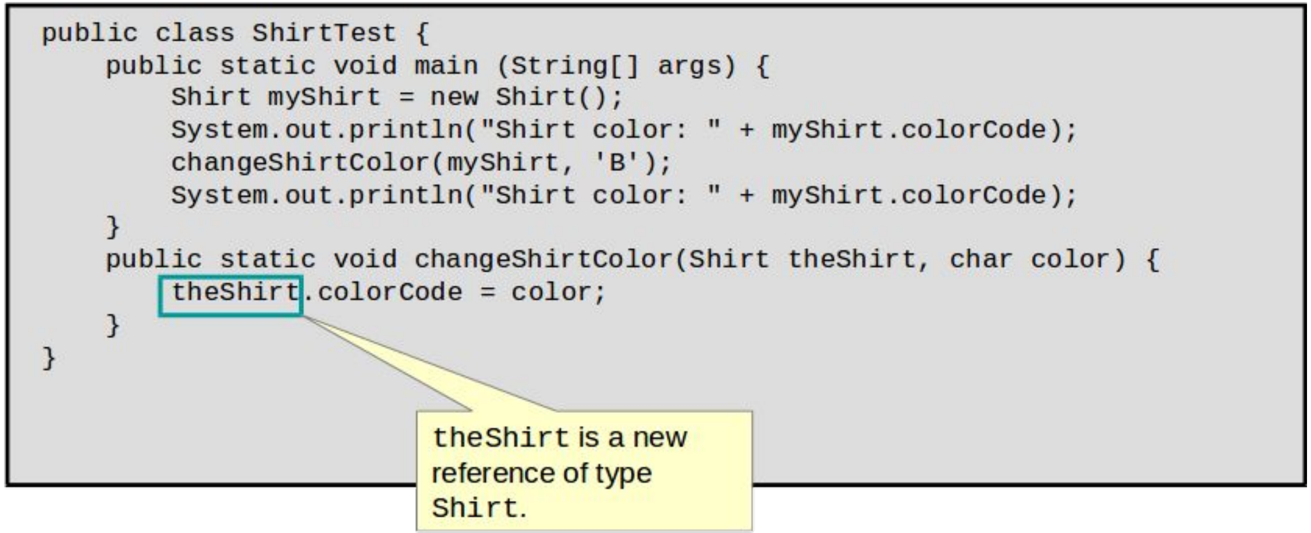
Invoking a Method in the Same Class



```
public class Elevator {

public boolean doorOpen=false;
public int currentFloor = 1;

public final int TOP_FLOOR = 5;
public final int BOTTOM_FLOOR = 1;

public void openDoor() {

           // Check if door already open
    if ( !isDoorOpen() ) {

                  // door opening code

    }
}
```

Evaluates to true if door is closed

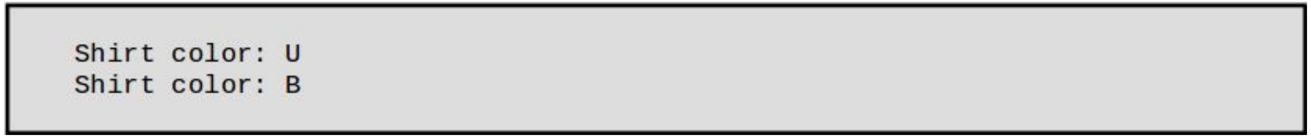
How Arguments Are Passed to Methods



```
public class ShirtTest {
    public static void main (String[] args) {
        Shirt myShirt = new Shirt();
        System.out.println("Shirt color: " + myShirt.colorCode);
        changeShirtColor(myShirt, 'B');
        System.out.println("Shirt color: " + myShirt.colorCode);
    }
    public static void changeShirtColor(Shirt theShirt, char color) {
        theShirt.colorCode = color;
    }
}
```

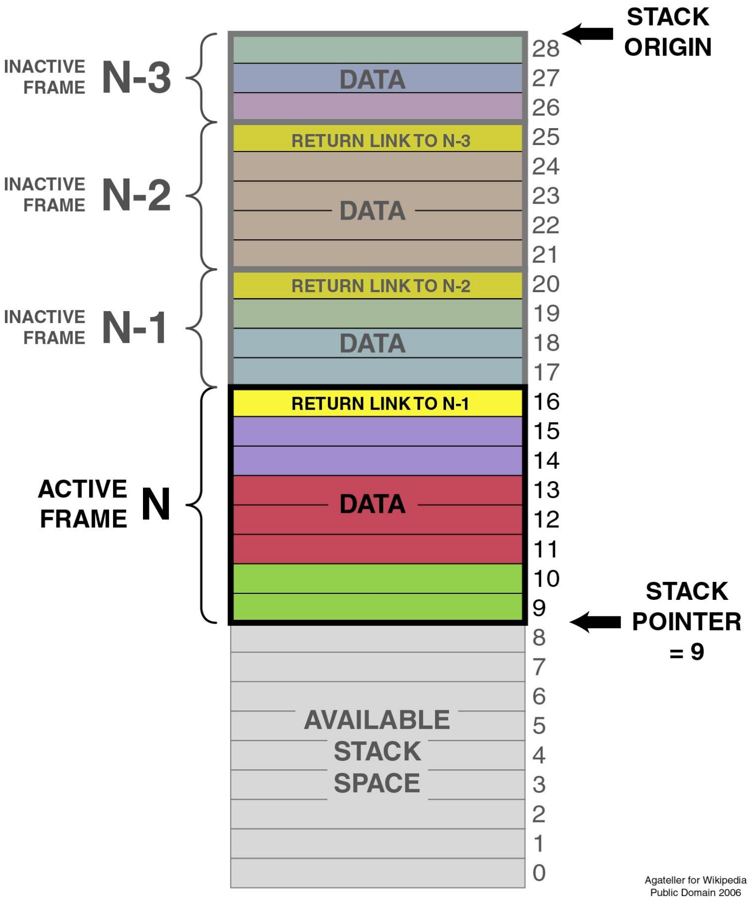
`theShirt` is a new reference of type `Shirt`.

Output:



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

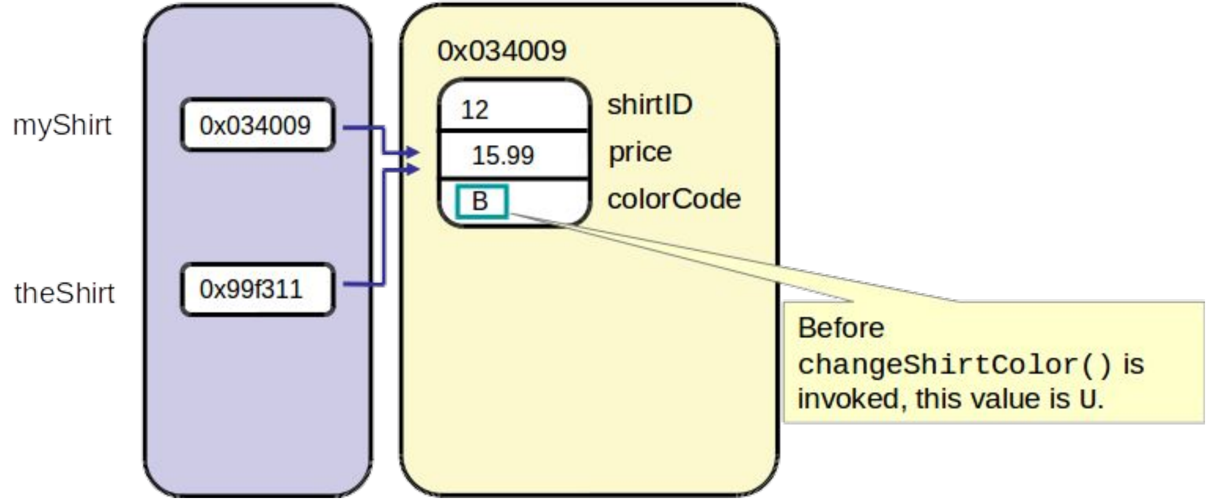
Stack



Agateller for Wikipedia
Public Domain 2006

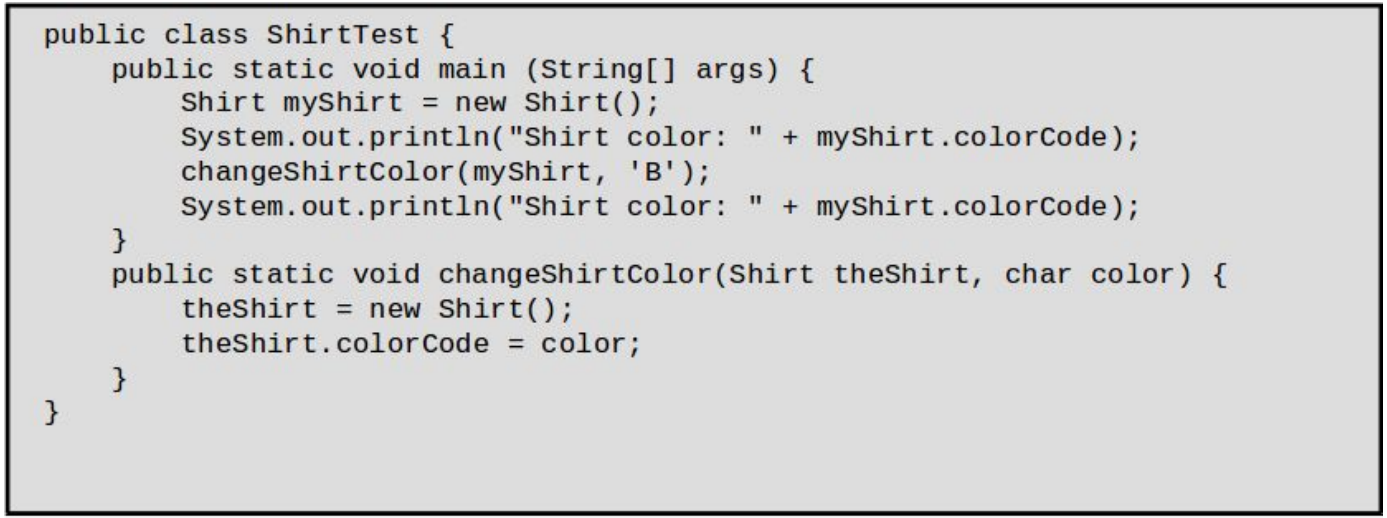
Passing by Value

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



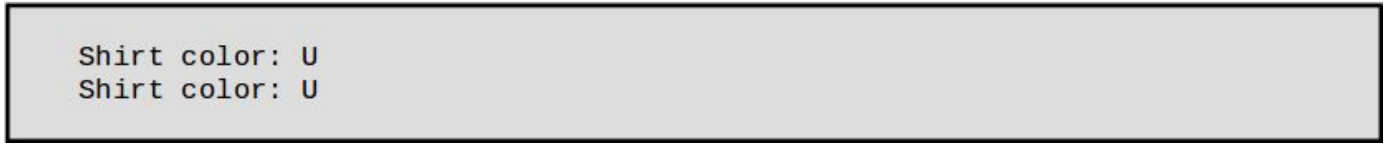
Before `changeShirtColor()` is invoked, this value is U.

Example



```
public class ShirtTest {
    public static void main (String[] args) {
        Shirt myShirt = new Shirt();
        System.out.println("Shirt color: " + myShirt.colorCode);
        changeShirtColor(myShirt, 'B');
        System.out.println("Shirt color: " + myShirt.colorCode);
    }
    public static void changeShirtColor(Shirt theShirt, char color) {
        theShirt = new Shirt();
        theShirt.colorCode = color;
    }
}
```

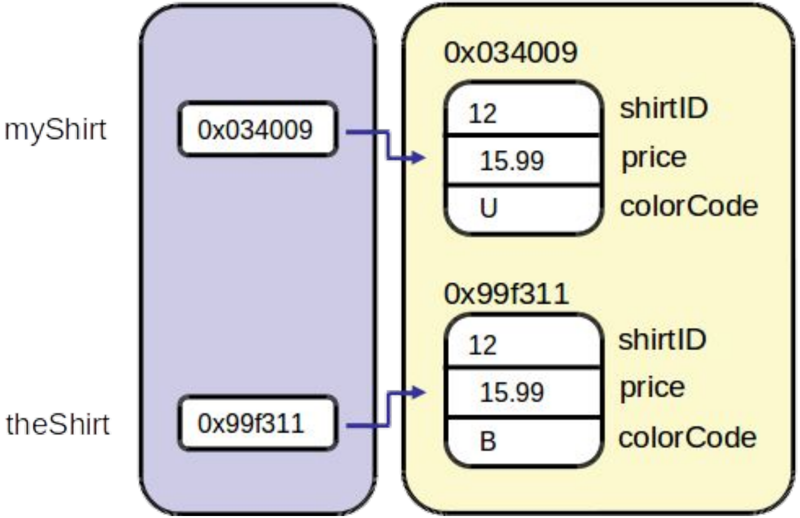
Output:



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

Example

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



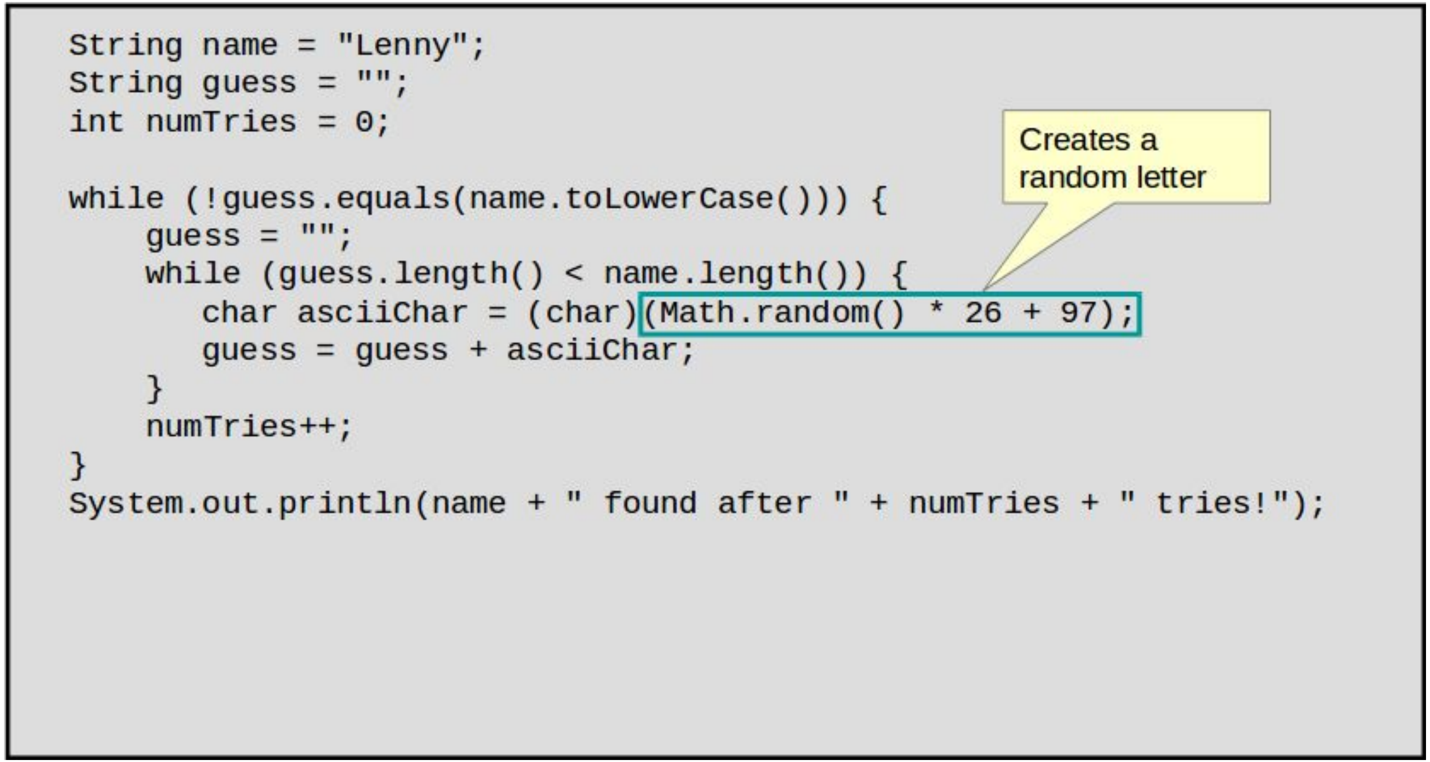
Advantages of Using Methods

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

Invoking Methods: Summary

- There is no limit to the number of method calls that a calling method can make
- The calling method and the worker method can be in the same class or in different classes
- You can invoke methods in any order
- Methods do not need to be completed in the order in which they are listed in the class where they are declared (the class containing the worker methods)
- All arguments passed into a method are passed by value

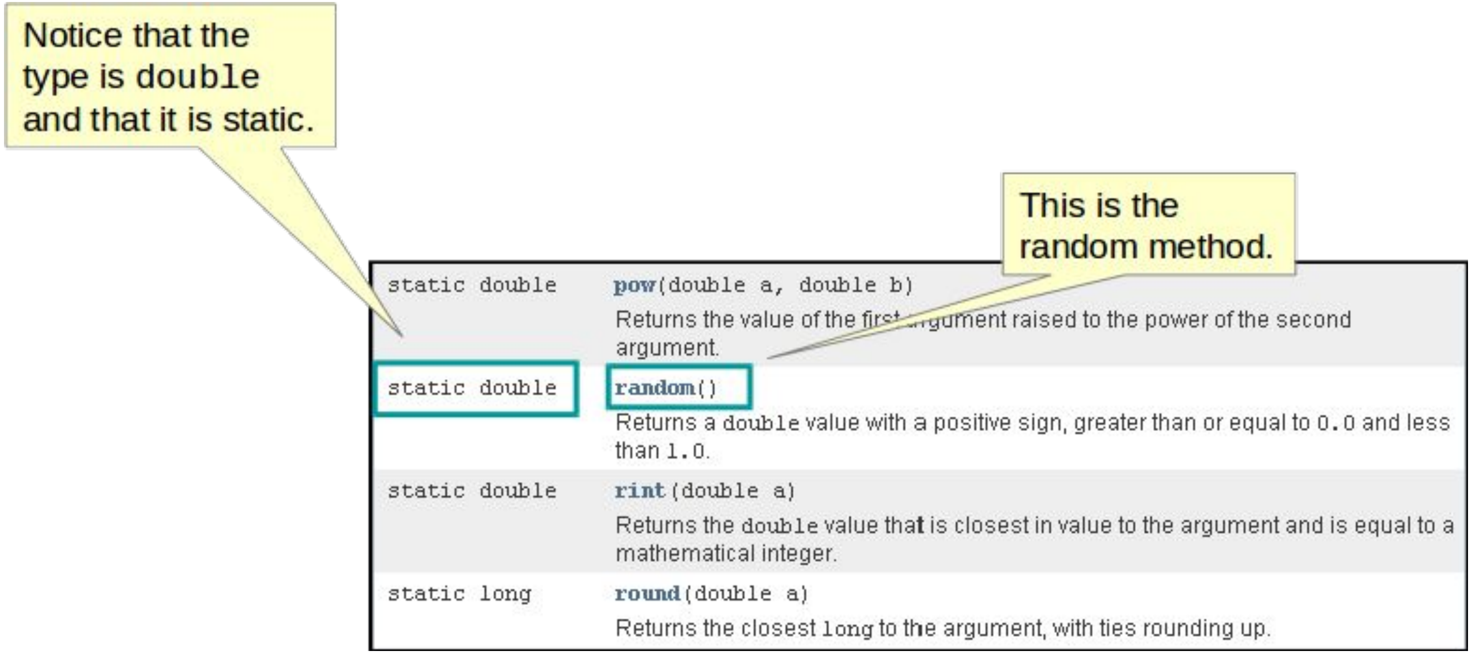
Math Utilities



```
String name = "Lenny";
String guess = "";
int numTries = 0;

while (!guess.equals(name.toLowerCase())) {
    guess = "";
    while (guess.length() < name.length()) {
       char asciiChar = (char)(Math.random() * 26 + 97);
       guess = guess + asciiChar;
    }
    numTries++;
}
System.out.println(name + " found after " + numTries + " tries!");
```

Creates a random letter

Static Methods in Math



Notice that the type is double and that it is static.

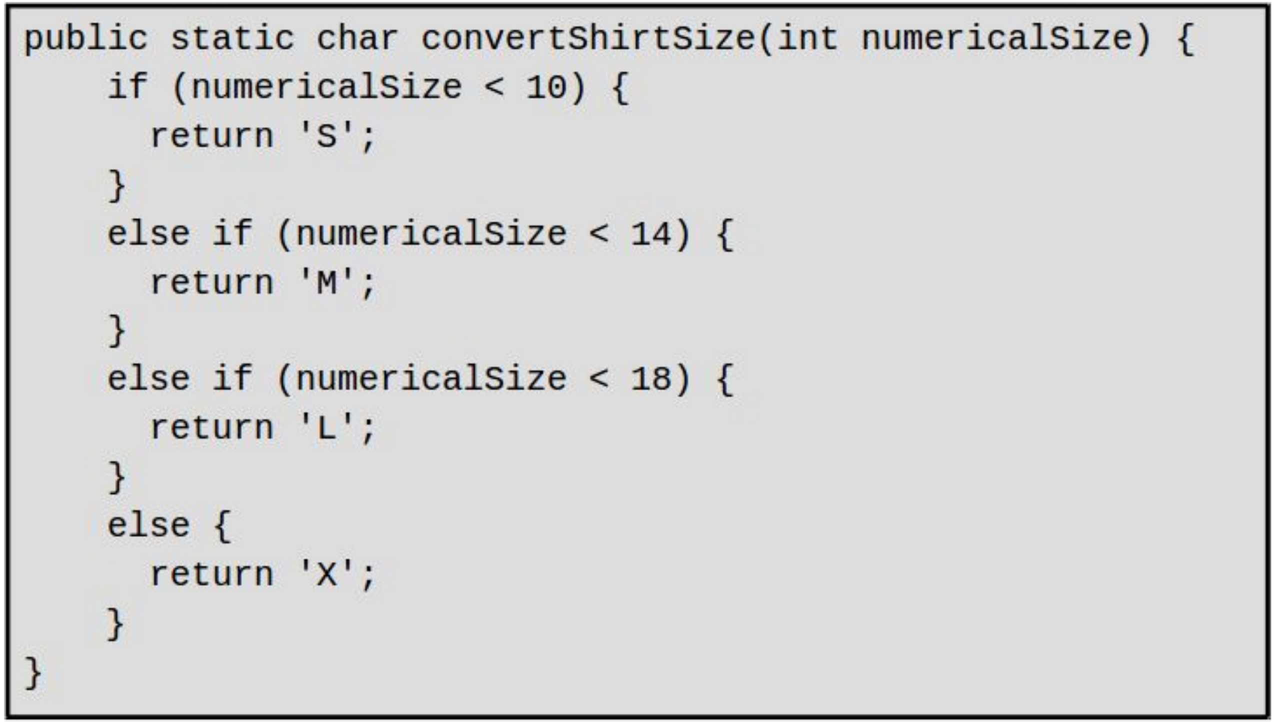
This is the random method.

`static double`	`pow(double a, double b)` Returns the value of the first argument raised to the power of the second argument.
`static double`	`random()` Returns a `double` value with a positive sign, greater than or equal to `0.0` and less than `1.0`.
`static double`	`rint(double a)` Returns the `double` value that is closest in value to the argument and is equal to a mathematical integer.
`static long`	`round(double a)` Returns the closest `long` to the argument, with ties rounding up.

Creating static Methods and Variables

- Methods and nonlocal variables can be static
- They belong to the class and not to the object
- They are declared using the static keyword:
    - *static Properties getProperties()*
- To invoke static methods:
    - *Classname.method();*
- To access static variables in another class:
    - *Classname.attribute_name;*
- To access static variables in the same class:
    - *attribute_name;*

Creating static Methods and Variables



```
public static char convertShirtSize(int numericalSize) {
    if (numericalSize < 10) {
      return 'S';
    }
    else if (numericalSize < 14) {
      return 'M';
    }
    else if (numericalSize < 18) {
      return 'L';
    }
    else {
      return 'X';
    }
}
```

Static Variables

- Declaring static variables:
  - *static double salesTAX = 8.25;*
- Accessing static variables:
  - *Classname.variable;*
- Example:
  - *double myPI;*
  - *myPI = Math.PI;*

Example of static Methods and Variables in the Java API

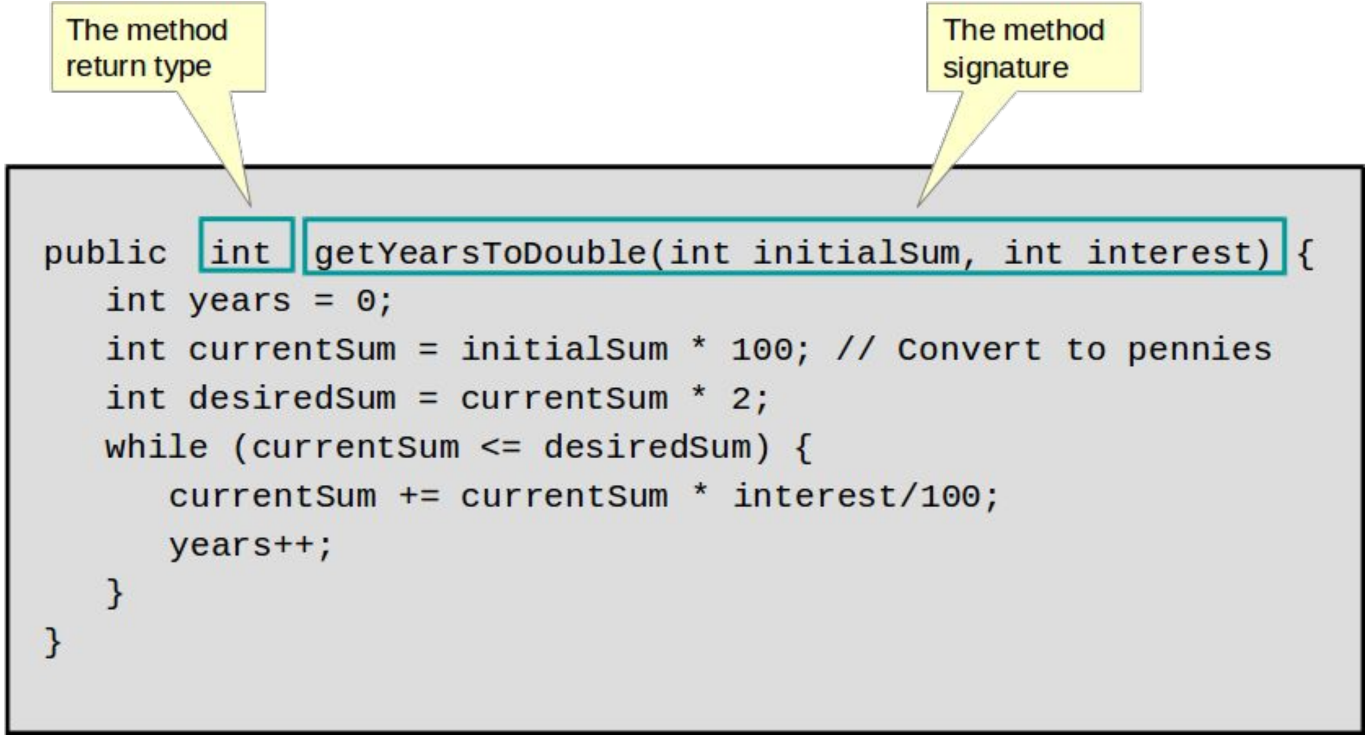
- Some functionality of the Math class:
  - Exponential
  - Logarithmic
  - Trigonometric
  - Random
  - Access to common mathematical constants, such as the value pi (Math.PI)
- Some functionality of the System class:
  - Retrieving environment variables
  - Access to the standard input and output streams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

Static Methods and Variables

When to declare a static method or variable:

- 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

Method Signature



The method return type

The method signature

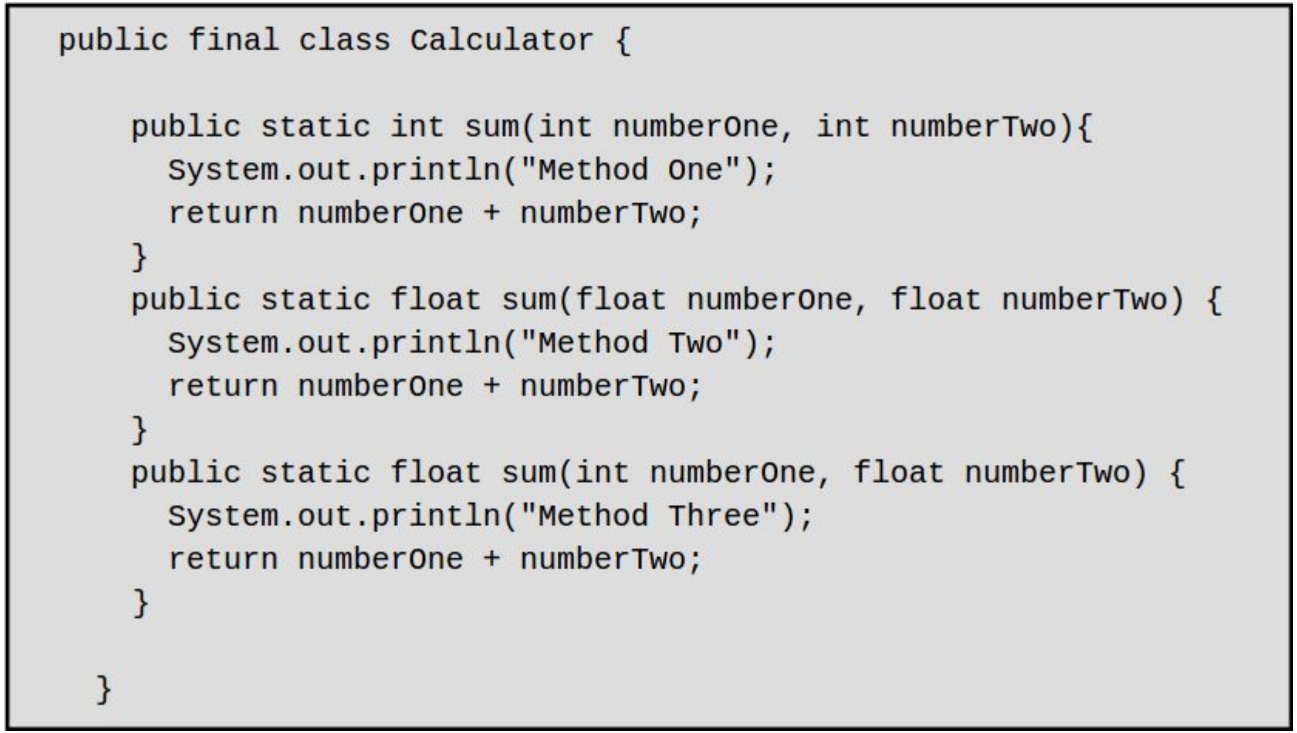
```
public  int  getYearsToDouble(int initialSum, int interest) {
   int years = 0;
   int currentSum = initialSum * 100; // Convert to pennies
   int desiredSum = currentSum * 2;
   while (currentSum <= desiredSum) {
      currentSum += currentSum * interest/100;
      years++;
   }
}
```

Method Overloading

Overloaded methods:

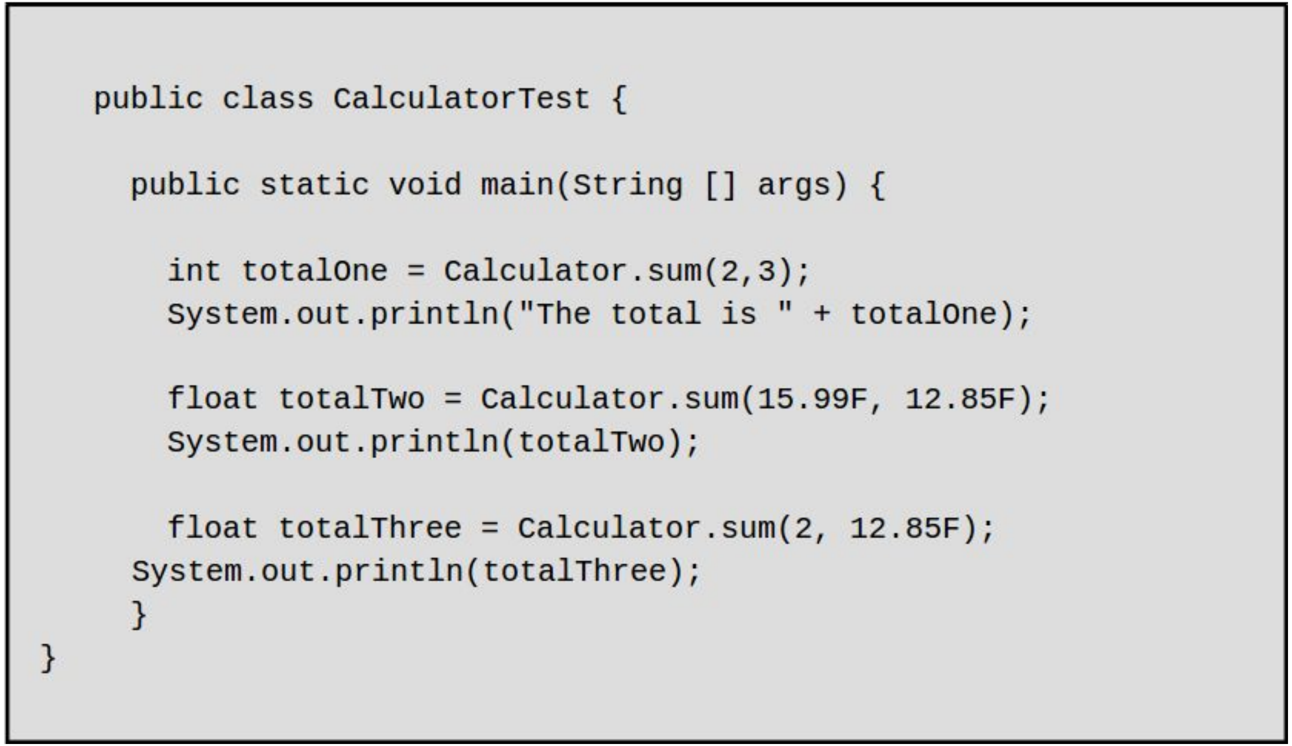
- Have the same name
- Have different signatures:
  - Different number and/or different type and/or different order of parameters
  - May have different functionality or similar functionality
- Are widely used in the foundation classes

Using Method Overloading



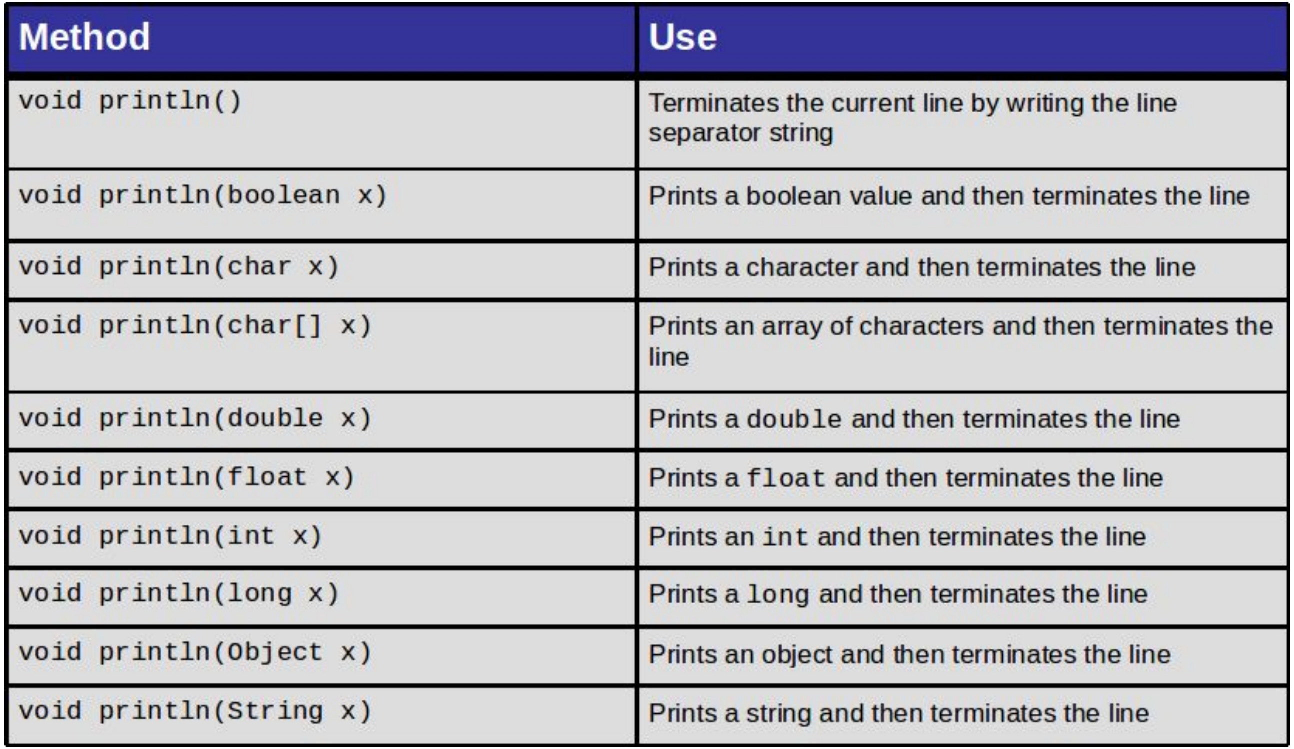
```
public final class Calculator {

    public static int sum(int numberOne, int numberTwo){
      System.out.println("Method One");
      return numberOne + numberTwo;
    }
    public static float sum(float numberOne, float numberTwo) {
      System.out.println("Method Two");
      return numberOne + numberTwo;
    }
    public static float sum(int numberOne, float numberTwo) {
      System.out.println("Method Three");
      return numberOne + numberTwo;
    }

  }
```

Using Method Overloading



```
  public class CalculatorTest {

    public static void main(String [] args) {

      int totalOne = Calculator.sum(2,3);
      System.out.println("The total is " + totalOne);

      float totalTwo = Calculator.sum(15.99F, 12.85F);
      System.out.println(totalTwo);

      float totalThree = Calculator.sum(2, 12.85F);
    System.out.println(totalThree);
    }
}
```

Method Overloading and the Java API

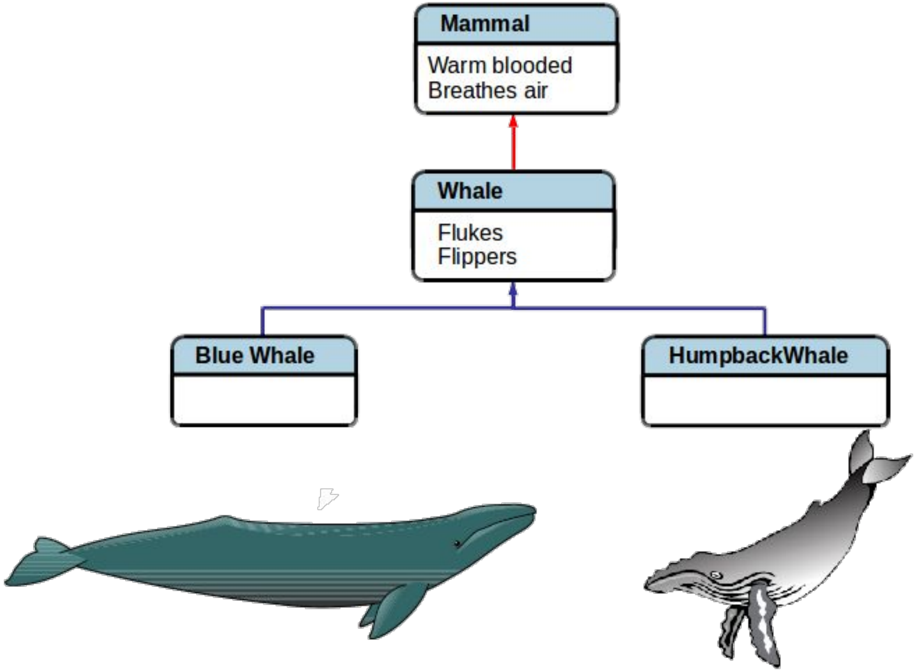


Method	Use
`void println()`	Terminates the current line by writing the line separator string
`void println(boolean x)`	Prints a boolean value and then terminates the line
`void println(char x)`	Prints a character and then terminates the line
`void println(char[] x)`	Prints an array of characters and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

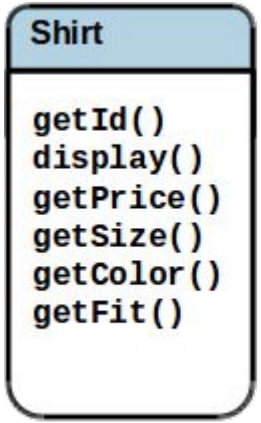
Object Oriented

- *Inheritance*
- *Abstract classes*
- *Polymorphism*
- *Interfaces*

Class Hierarchies

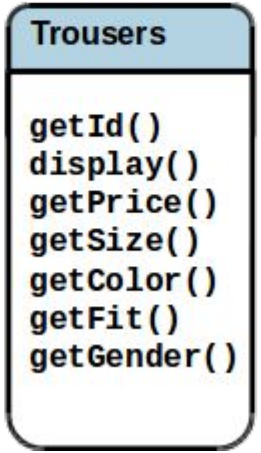


Code Duplication



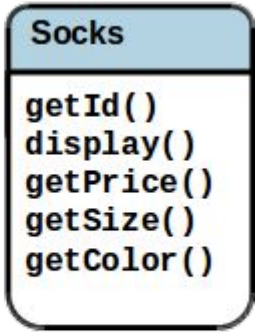
Shirt

```
getId()
display()
getPrice()
getSize()
getColor()
getFit()
```



Trousers

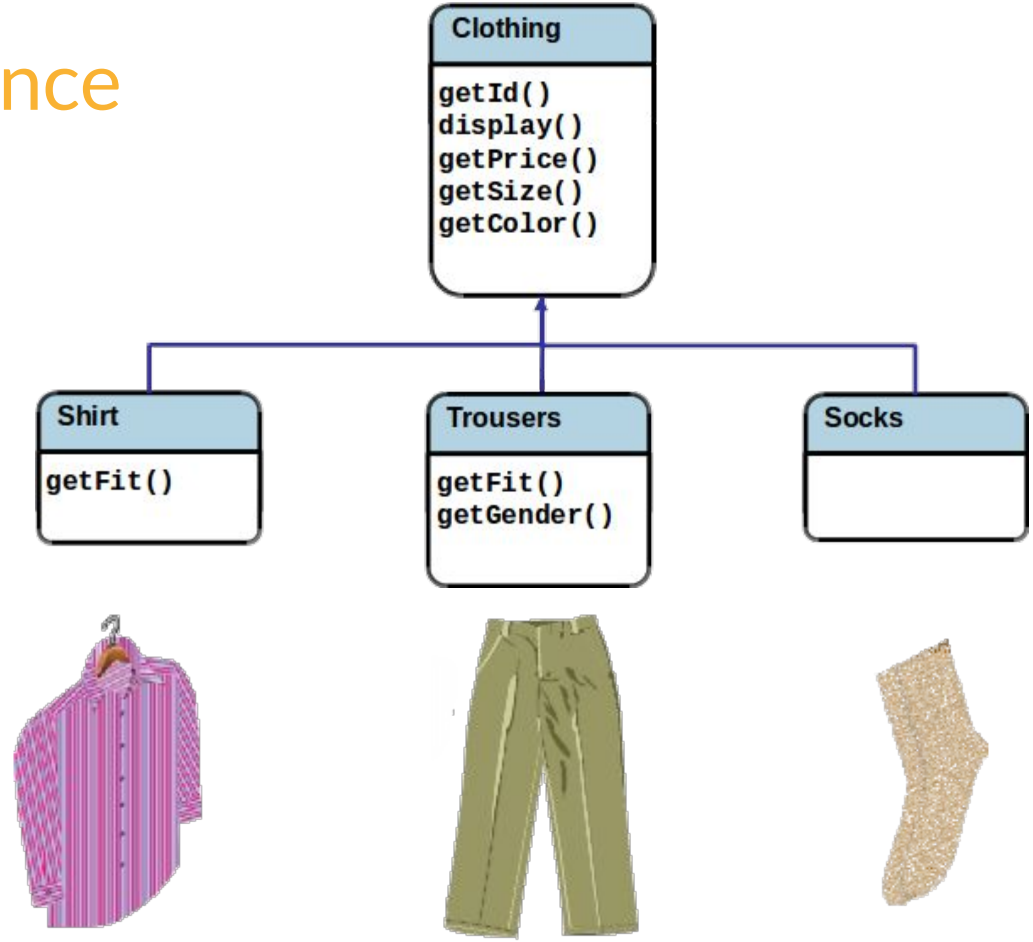
```
getId()
display()
getPrice()
getSize()
getColor()
getFit()
getGender()
```



Socks

```
getId()
display()
getPrice()
getSize()
getColor()
```

Inheritance



Clothing

getId()
display()
getPrice()
getSize()
getColor()

Shirt

getFit()

Trousers

getFit()
getGender()

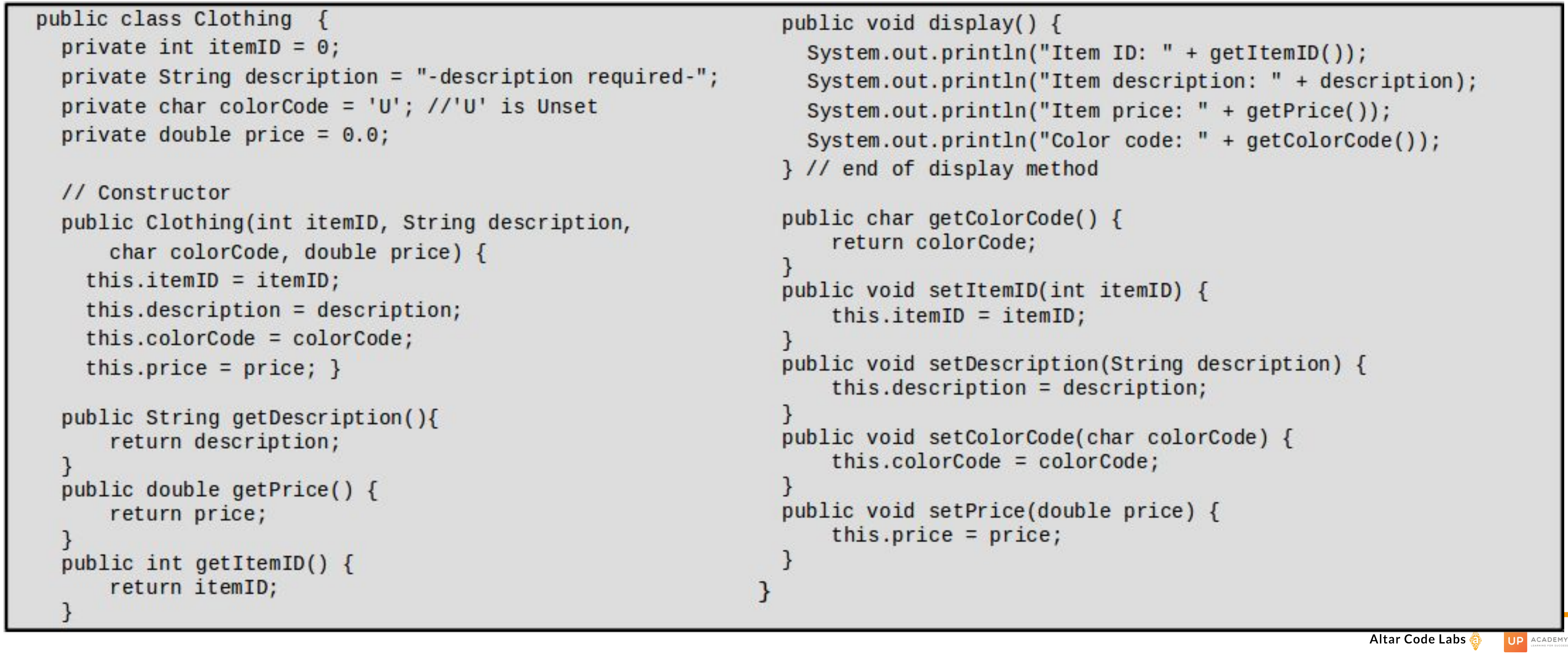
Socks

Overriding Superclass Methods

Methods that exist in the superclass can be:

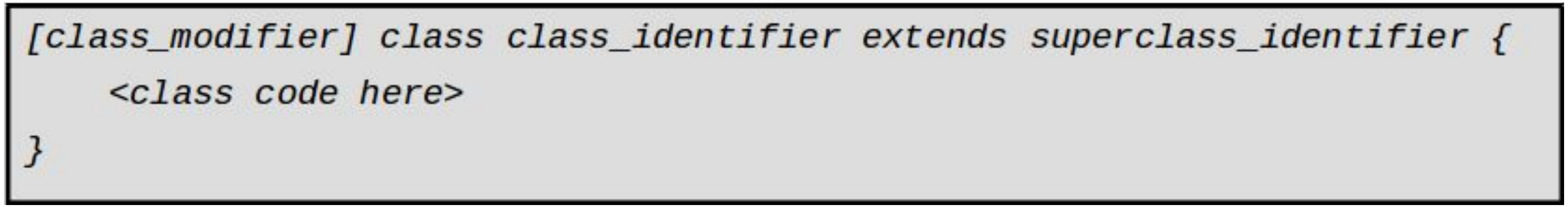
- Not implemented in the subclass:
  - The method declared in the superclass is used at runtime.
- Implemented in the subclass:
  - The method declared in the subclass is used at runtime.

Clothing Superclass



```java
public class Clothing  {
  private int itemID = 0;
  private String description = "-description required-";
  private char colorCode = 'U'; //'U' is Unset
  private double price = 0.0;

  // Constructor
  public Clothing(int itemID, String description,
      char colorCode, double price) {
    this.itemID = itemID;
    this.description = description;
    this.colorCode = colorCode;
    this.price = price; }

  public String getDescription(){
      return description;
  }
  public double getPrice() {
      return price;
  }
  public int getItemID() {
      return itemID;
  }
  public void display() {
    System.out.println("Item ID: " + getItemID());
    System.out.println("Item description: " + description);
    System.out.println("Item price: " + getPrice());
    System.out.println("Color code: " + getColorCode());
  } // end of display method

  public char getColorCode() {
      return colorCode;
  }
  public void setItemID(int itemID) {
      this.itemID = itemID;
  }
  public void setDescription(String description) {
      this.description = description;
  }
  public void setColorCode(char colorCode) {
      this.colorCode = colorCode;
  }
  public void setPrice(double price) {
      this.price = price;
  }
}
```

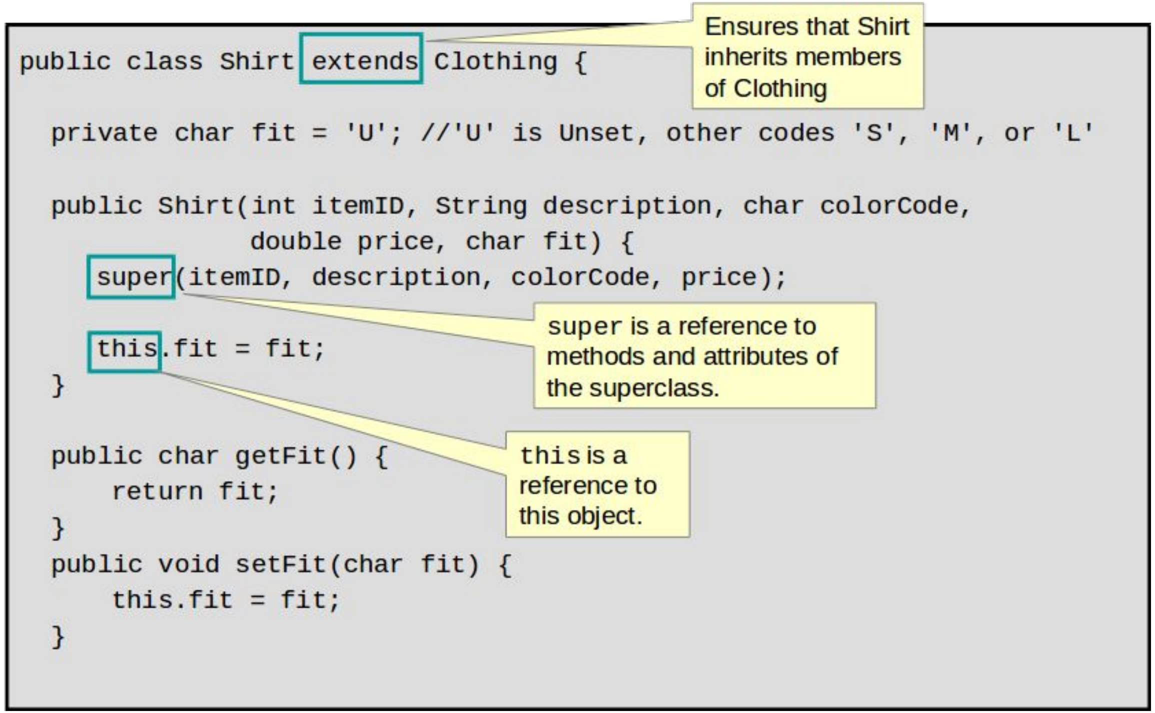
Declaring a Subclass



Syntax:

```
[class_modifier] class class_identifier extends superclass_identifier {
    <class code here>
}
```

Declaring a Subclass (extends, super and this keywords)



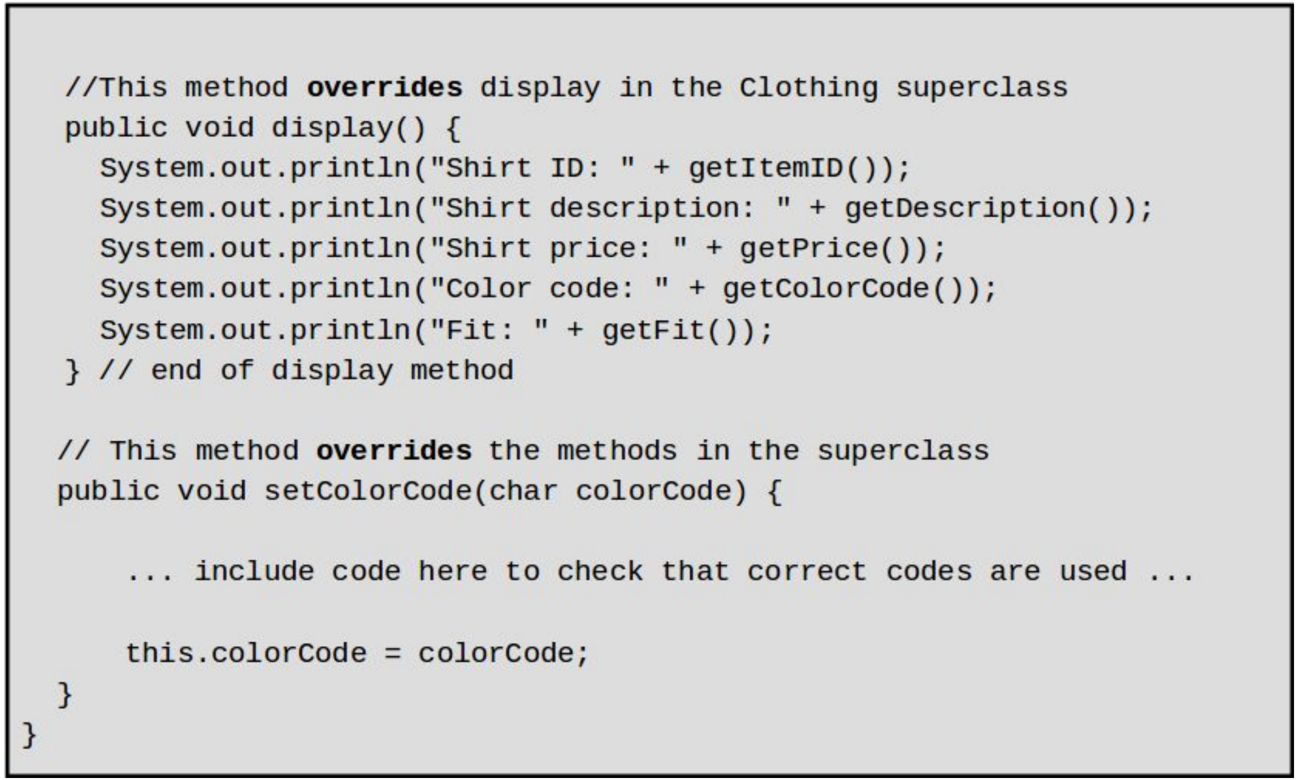
```
public class Shirt extends Clothing {

  private char fit = 'U'; //'U' is Unset, other codes 'S', 'M', or 'L'

  public Shirt(int itemID, String description, char colorCode,
               double price, char fit) {
    super(itemID, description, colorCode, price);

    this.fit = fit;
  }

  public char getFit() {
      return fit;
  }
  public void setFit(char fit) {
      this.fit = fit;
  }
```

Ensures that Shirt inherits members of Clothing

super is a reference to methods and attributes of the superclass.

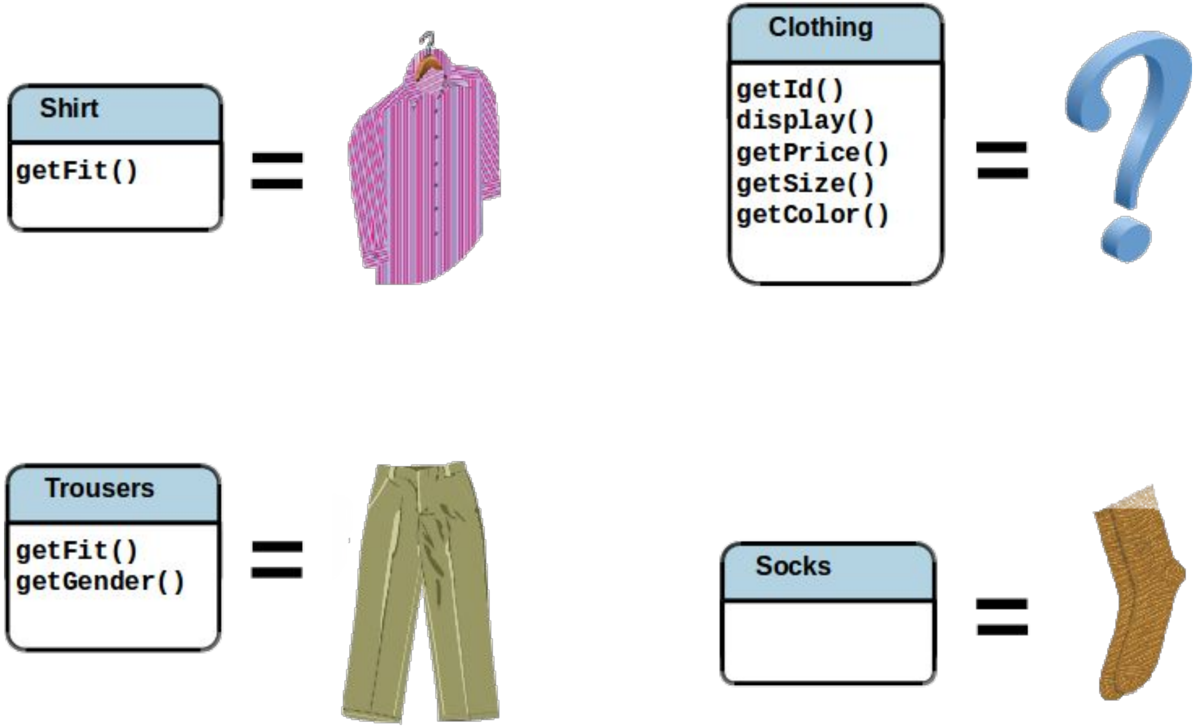
this is a reference to this object.

Declaring a Subclass (overriding)



```
  //This method overrides display in the Clothing superclass
  public void display() {
    System.out.println("Shirt ID: " + getItemID());
    System.out.println("Shirt description: " + getDescription());
    System.out.println("Shirt price: " + getPrice());
    System.out.println("Color code: " + getColorCode());
    System.out.println("Fit: " + getFit());
  } // end of display method

  // This method overrides the methods in the superclass
  public void setColorCode(char colorCode) {

      ... include code here to check that correct codes are used ...

      this.colorCode = colorCode;
  }
}
```

Abstract Classes



Shirt

getFit()

=

Clothing

getId()
display()
getPrice()
getSize()
getColor()

=

Trousers

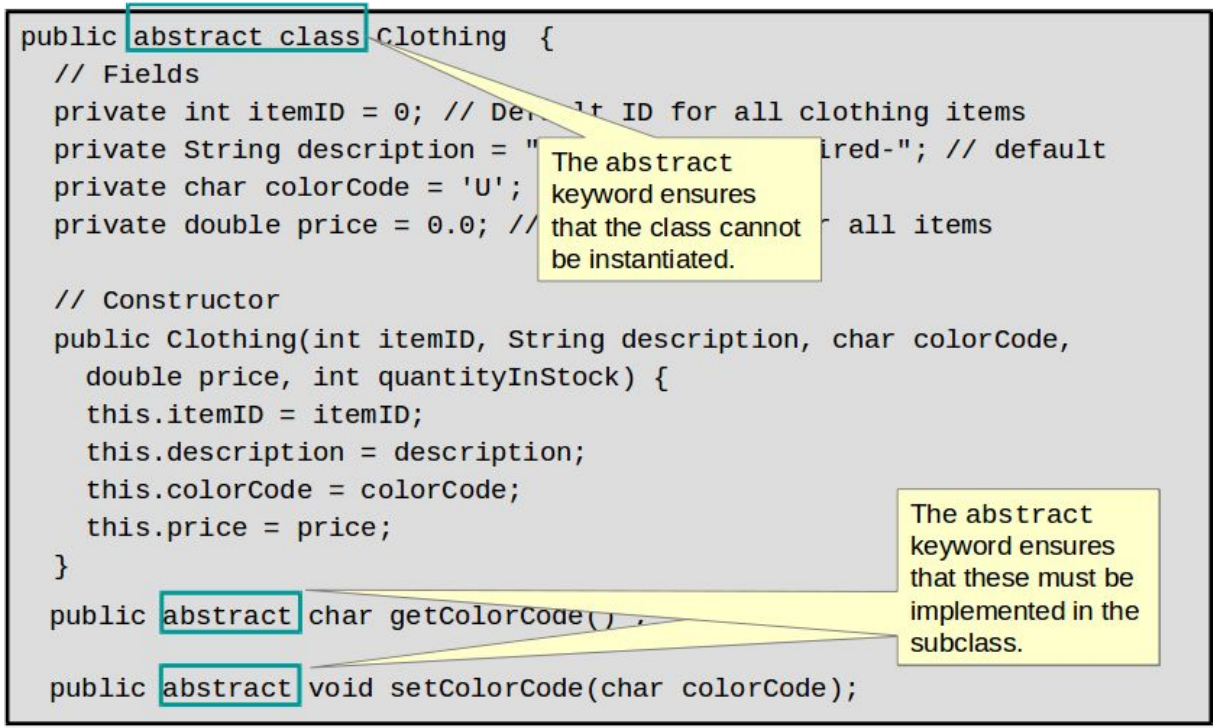
getFit()
getGender()

=

Socks

=

Abstract Clothing Superclass



```
public abstract class Clothing  {
  // Fields
  private int itemID = 0; // De    lt ID for all clothing items
  private String description = "              ired-"; // default
  private char colorCode = 'U';
  private double price = 0.0; //               r all items

  // Constructor
  public Clothing(int itemID, String description, char colorCode,
    double price, int quantityInStock) {
    this.itemID = itemID;
    this.description = description;
    this.colorCode = colorCode;
    this.price = price;
  }
  public abstract char getColorCode();

  public abstract void setColorCode(char colorCode);
```

The abstract keyword ensures that the class cannot be instantiated.

The abstract keyword ensures that these must be implemented in the subclass.

Superclass and Subclass Relationships

It is very important to consider the best use of inheritance:

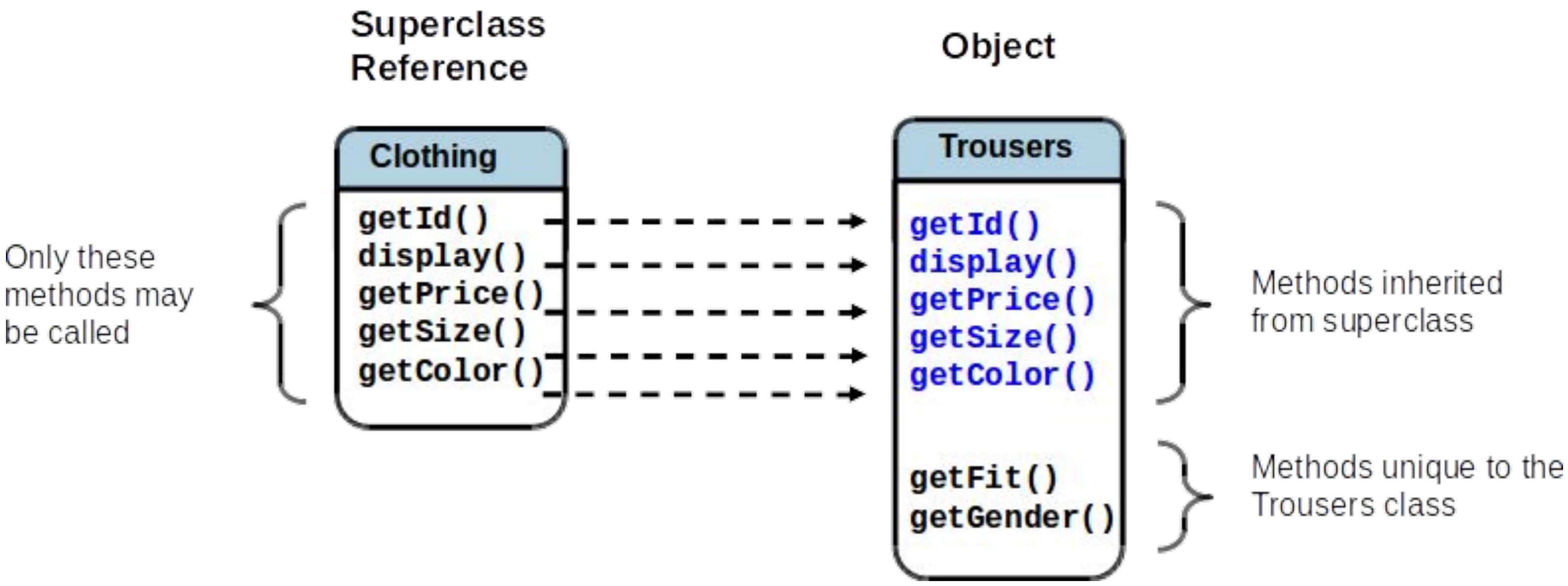
- **Use inheritance only** when it is completely **valid or unavoidable**
- Check appropriateness with the **“is a” phrase**:
  - The phrase “a Shirt **is a** piece of Clothing” expresses a **valid** inheritance link.
  - The phrase “a Hat **is a** Sock” expresses an **invalid** inheritance link.

Superclass Reference Types

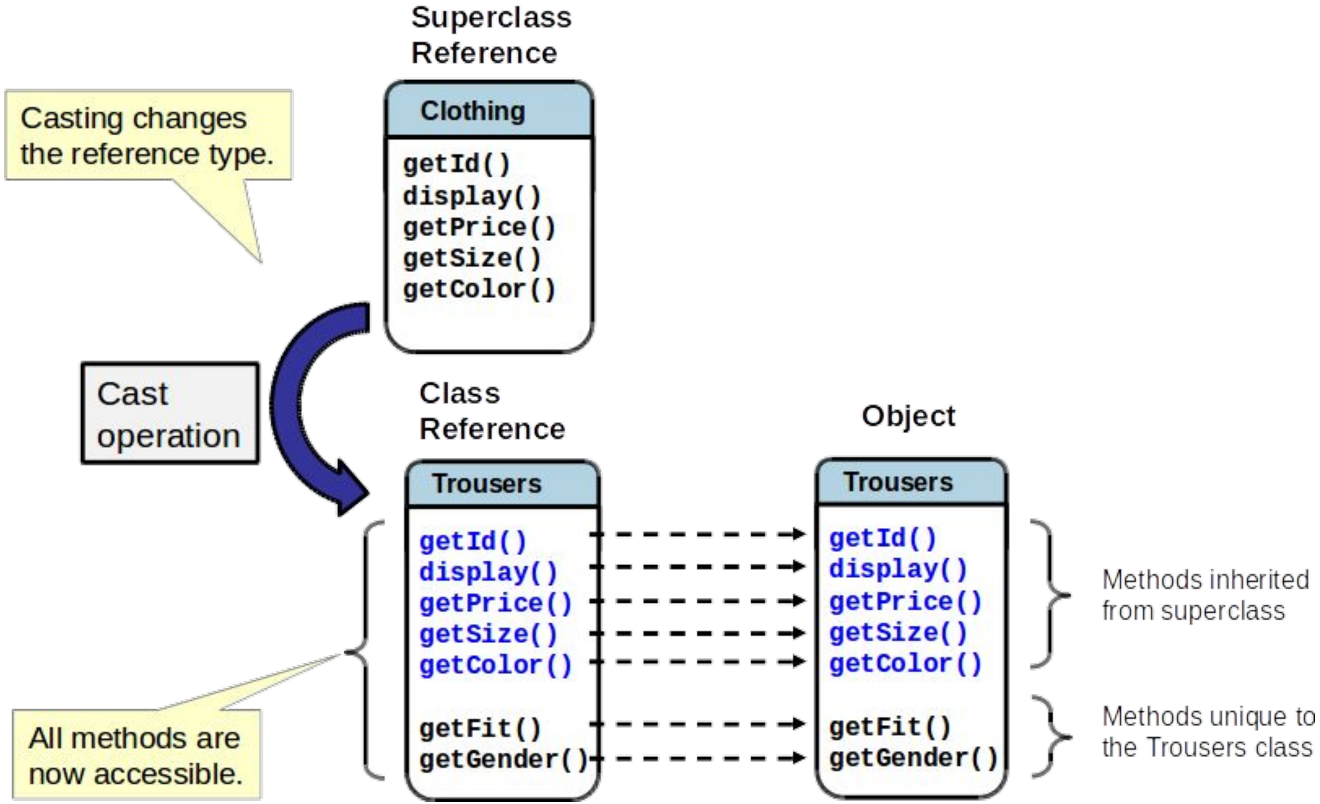
So far we have seen the class used as the reference type for the created object:

- To use the Shirt class as the reference type for the Shirt object:
  - Shirt myShirt = new Shirt();
- But we can also use the superclass as the reference:
  - Clothing clothingItem1 = new Shirt();
  - Clothing clothingItem2 = new Trousers();

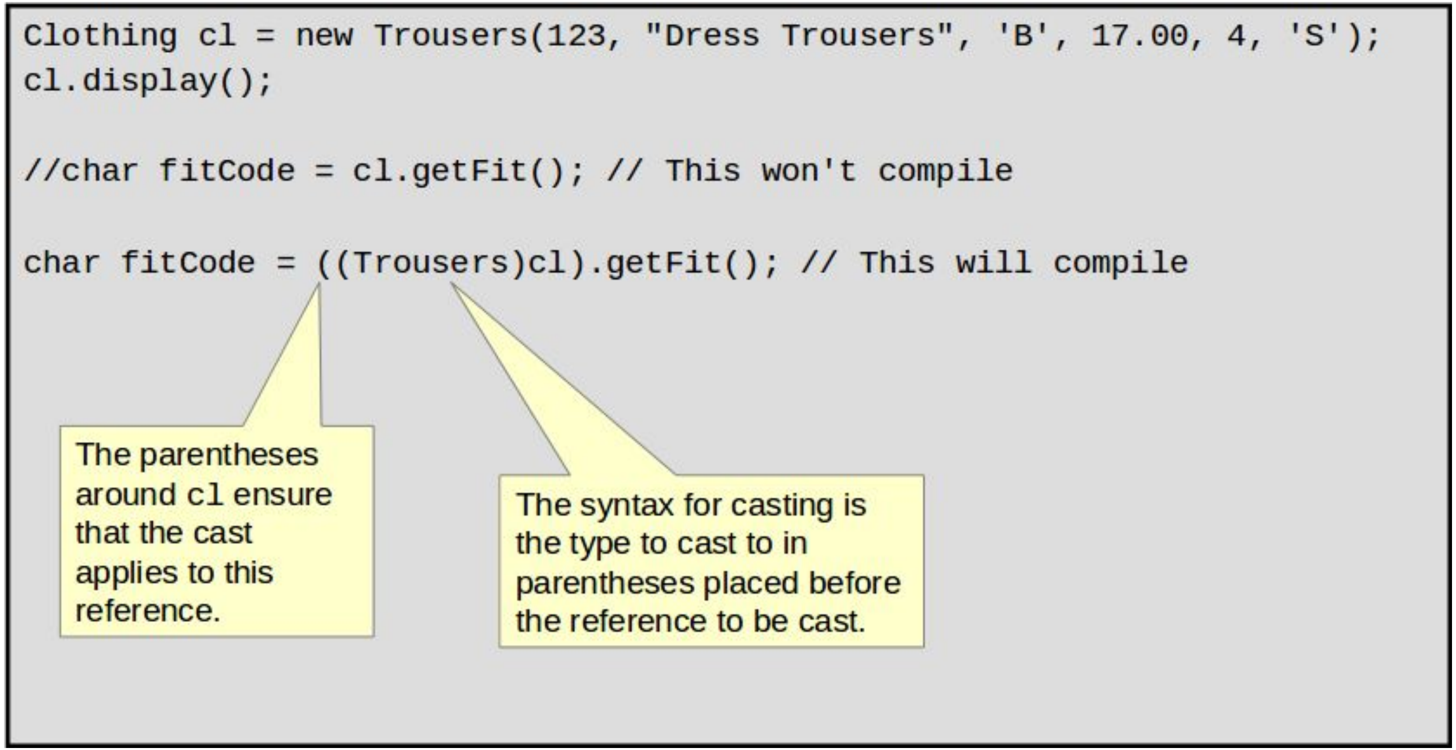
Accessing Class Methods from Superclass



Casting the Reference Type



Casting



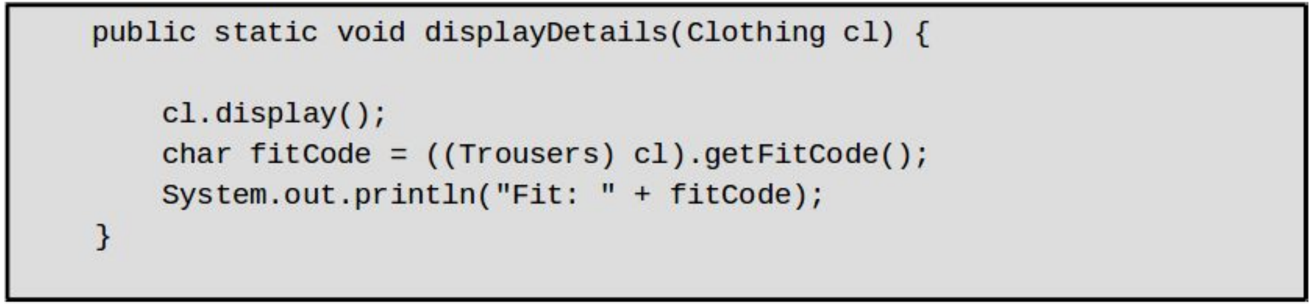
```
Clothing cl = new Trousers(123, "Dress Trousers", 'B', 17.00, 4, 'S');
cl.display();

//char fitCode = cl.getFit(); // This won't compile

char fitCode = ((Trousers)cl).getFit(); // This will compile
```

The parentheses around `cl` ensure that the cast applies to this reference.

The syntax for casting is the type to cast to in parentheses placed before the reference to be cast.

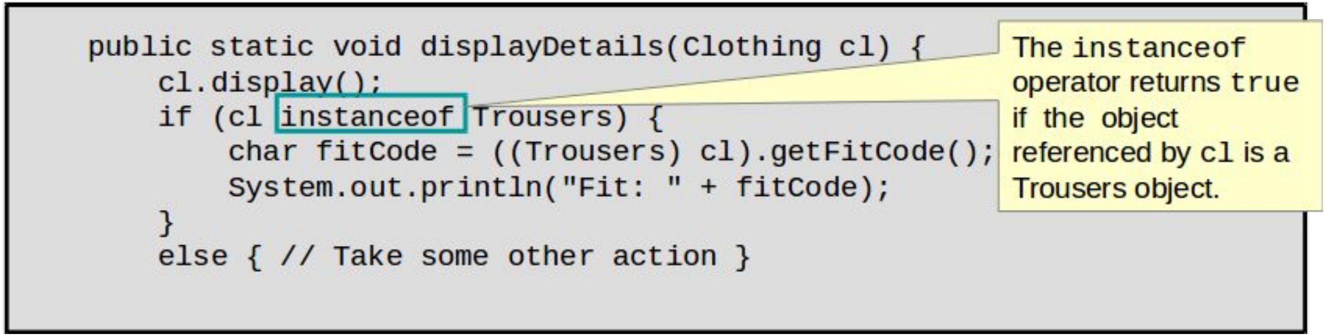
instanceof Operator

Possible casting error:



```
    public static void displayDetails(Clothing cl) {

        cl.display();
        char fitCode = ((Trousers) cl).getFitCode();
        System.out.println("Fit: " + fitCode);
    }
```

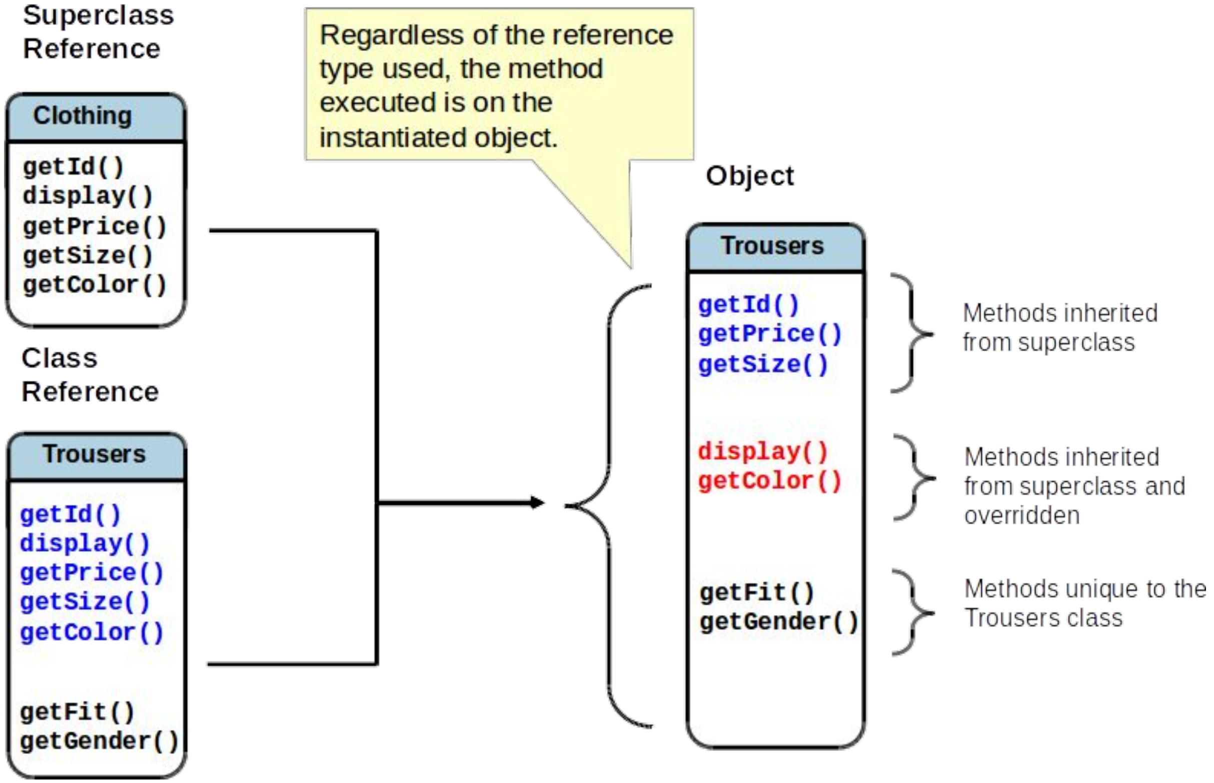
`instanceof` operator used to ensure there is no casting error:



```
    public static void displayDetails(Clothing cl) {
        cl.display();
        if (cl instanceof Trousers) {
            char fitCode = ((Trousers) cl).getFitCode();
            System.out.println("Fit: " + fitCode);
        }
        else { // Take some other action }
```

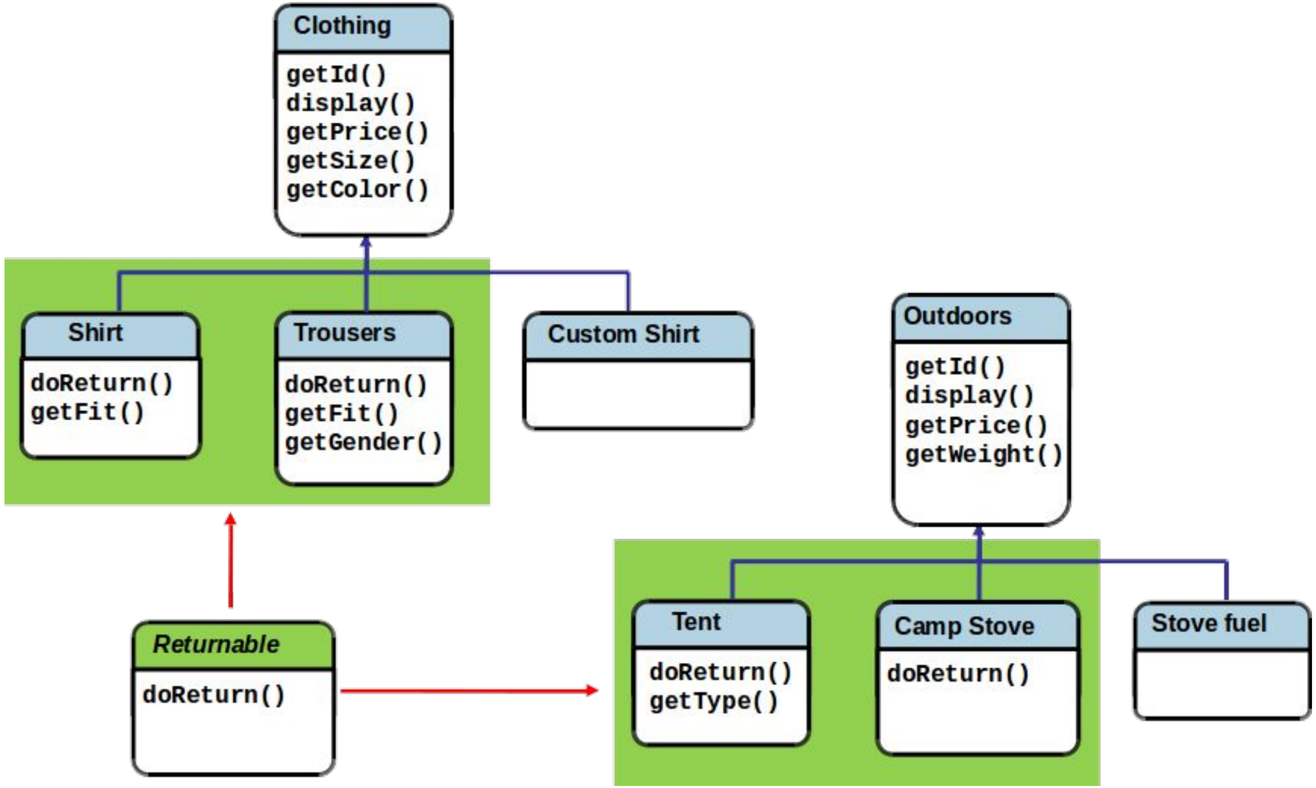
The `instanceof` operator returns `true` if the object referenced by `cl` is a Trousers object.

Polymorphic Method Calls

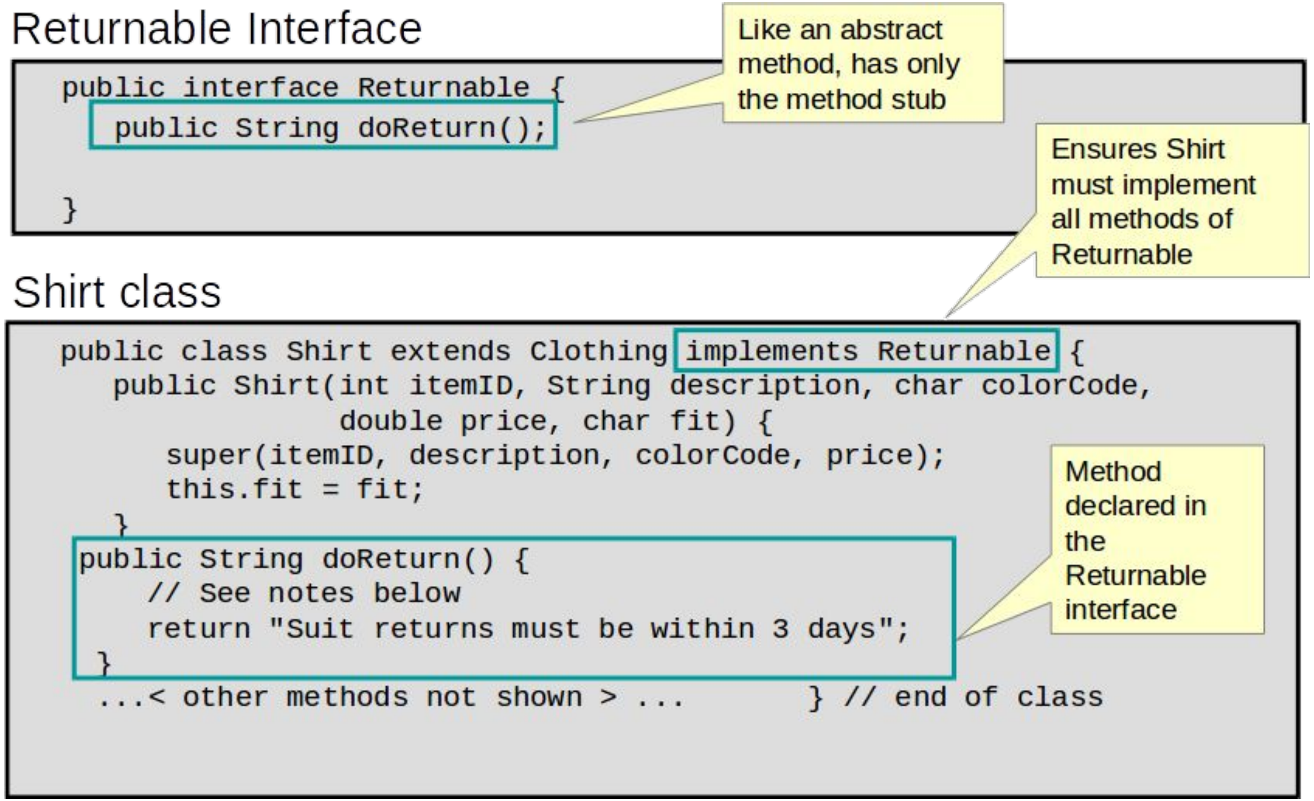


Regardless of the reference type used, the method executed is on the instantiated object.

Interfaces


Clothing
getId()
display()
getPrice()
getSize()
getColor()
Shirt
doReturn()
getFit()
Trousers
doReturn()
getFit()
getGender()
Custom Shirt
Outdoors
getId()
display()
getPrice()
getWeight()
Returnable
doReturn()
Tent
doReturn()
getType()
Camp Stove
doReturn()
Stove fuel

Implementing the Returnable Interface



Returnable Interface

```
public interface Returnable {
    public String doReturn();

}
```

Like an abstract method, has only the method stub

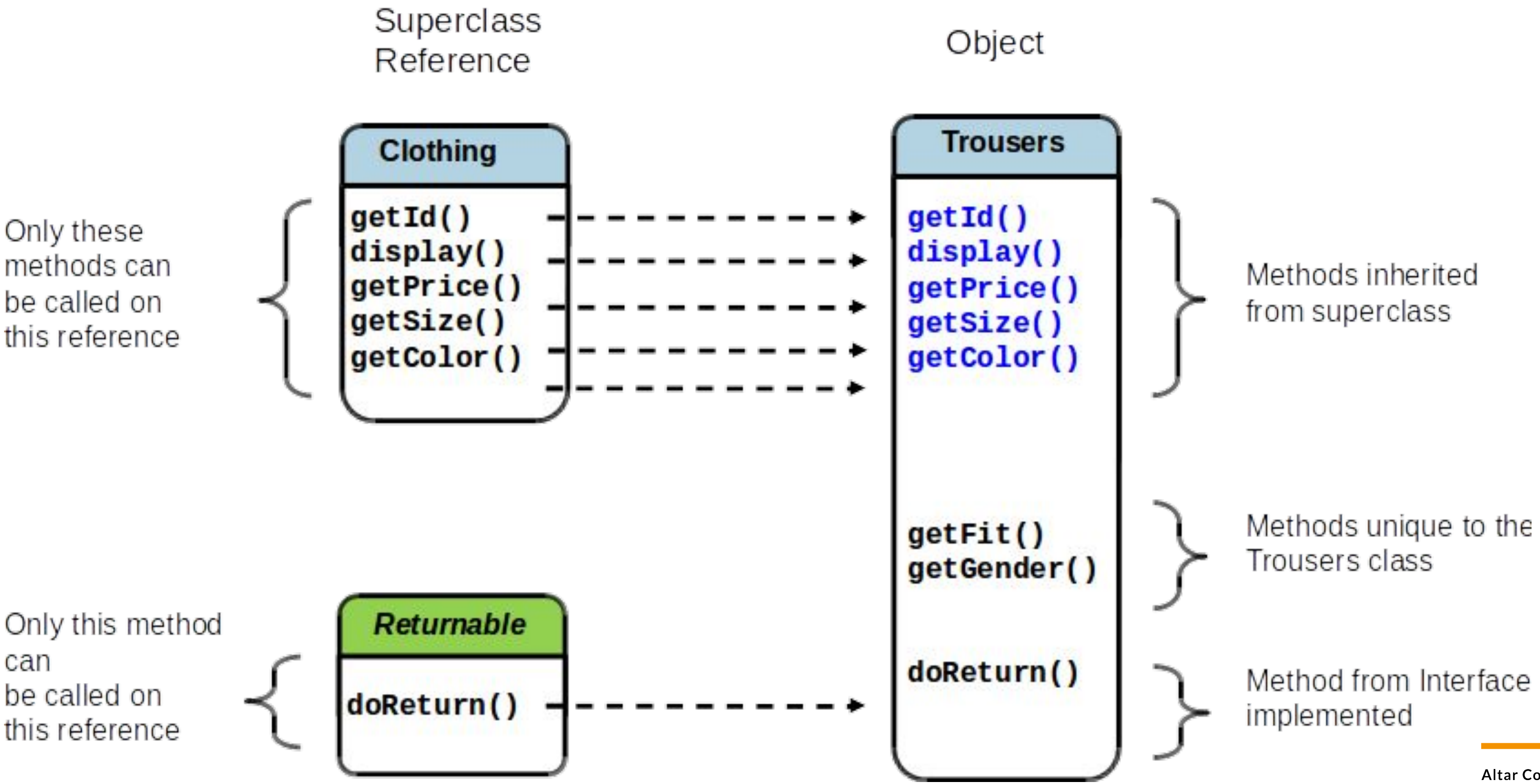
Ensures Shirt must implement all methods of Returnable

Shirt class

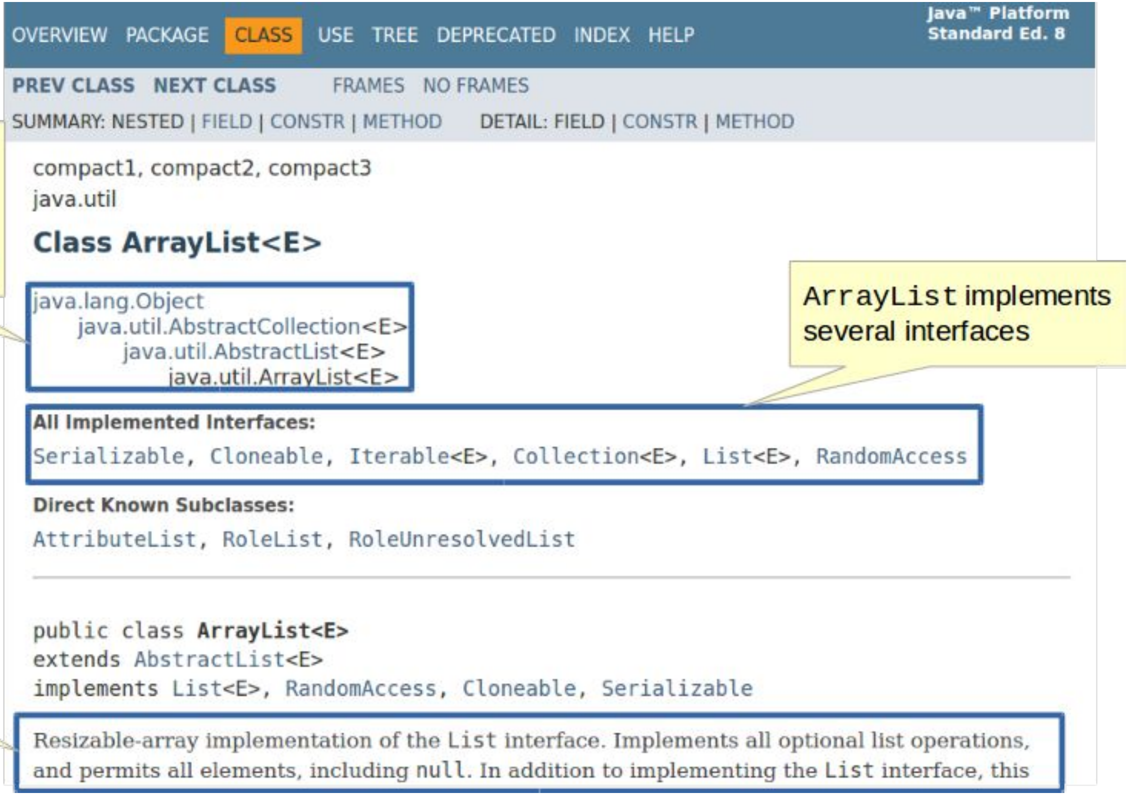
```
public class Shirt extends Clothing implements Returnable {
   public Shirt(int itemID, String description, char colorCode,
                double price, char fit) {
      super(itemID, description, colorCode, price);
      this.fit = fit;
   }
  public String doReturn() {
     // See notes below
     return "Suit returns must be within 3 days";
  }
  ...< other methods not shown > ...      } // end of class
```

Method declared in the Returnable interface

Access to Object Methods from Interface


Superclass Reference
Object
Clothing
getId()
display()
getPrice()
getSize()
getColor()
Trousers
getId()
display()
getPrice()
getSize()
getColor()
getFit()
getGender()
doReturn()
Returnable
doReturn()
Only these methods can be called on this reference
Methods inherited from superclass
Methods unique to the Trousers class
Only this method can be called on this reference
Method from Interface implemented

ArrayList example


OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
Java™ Platform
Standard Ed. 8
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3
java.util

Class ArrayList<E>

```
java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.ArrayList<E>
```

All Implemented Interfaces:

`Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess`

Direct Known Subclasses:

`AttributeList, RoleList, RoleUnresolvedList`

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to implementing the `List` interface, this

`ArrayList` extends `AbstractList`, which extends `AbstractCollection`.

`ArrayList` implements several interfaces

The most used methods of this class are the ones declared on List interface

Luís Ribeiro

I'm a software engineer with more than 14 years of experience in software development in Java and other technologies, software architecture, team management and project management.

My mission is to transform people's ideas into fully functional, production ready and user friendly software applications that can change the world!

luismmribeiro@gmail.com

Thank you