

Recap of OO concepts

Objects, classes, methods and more.

Produced Dr. Siobhán Drohan
by: Mr. Colm Dunphy
 Mr. Diarmuid O'Connor
 Dr. Frank Walsh



Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
<http://www.wit.ie/>

Classes and Objects

- A **class**
 - defines a group of related **methods** (functions) and **fields** (variables / properties).

The screenshot shows the Oracle Java API documentation for the `String` class. The browser address bar shows the URL `https://docs.oracle.com/javase/7/docs/api`. The navigation menu includes `Overview`, `Package`, `Class` (selected), `Use`, `Tree`, `Deprecated`, `Index`, and `Help`. Below the navigation menu, there are links for `Prev Class`, `Next Class`, `Frames`, `No Frames`, and `All Classes`. The main content area is divided into two sections: a left sidebar and a right main area.

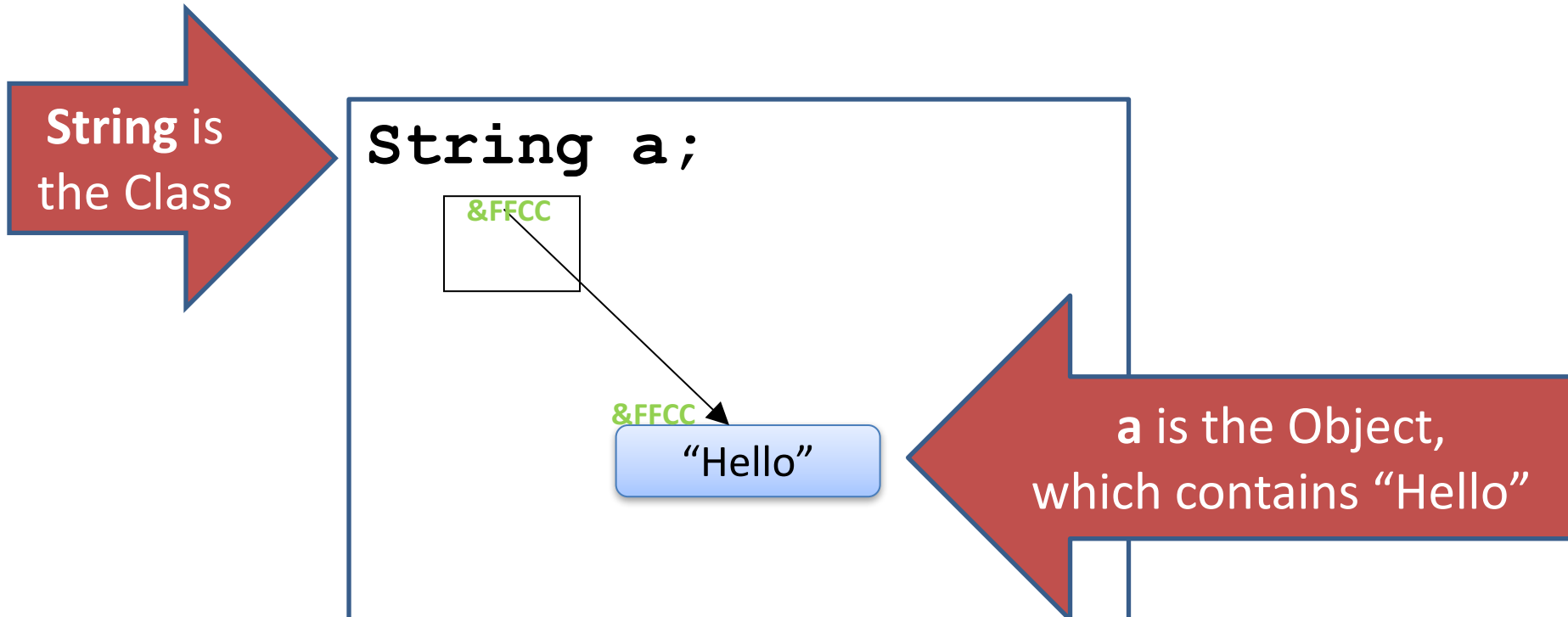
The left sidebar shows the package hierarchy: `java.lang`, `Class String`, `java.lang.Object`, and `java.lang.String`.

The right main area is titled `Method Summary` and contains a table of methods. The table has two columns: `Modifier and Type` and `Method and Description`.

Modifier and Type	Method and Description
char	<code>charAt(int index)</code> Returns the char value at the specified index.
int	<code>codePointAt(int index)</code> Returns the character (Unicode code point) at the specified index.
int	<code>codePointBefore(int index)</code> Returns the character (Unicode code point) before the specified index.
int	<code>codePointCount(int beginIndex, int endIndex)</code> Returns the number of Unicode code points in the specified text range of this <code>String</code> .
int	<code>compareTo(String anotherString)</code> Compares two strings lexicographically.
int	<code>compareToIgnoreCase(String str)</code> Compares two strings lexicographically, ignoring case differences.
<code>String</code>	<code>concat(String str)</code> Concatenates the specified string to the end of this string.
boolean	<code>contains(CharSequence s)</code> Returns true if and only if this string contains the specified sequence of char values.
boolean	<code>contentEquals(CharSequence cs)</code> Compares this string to the specified <code>CharSequence</code> .
boolean	<code>contentEquals(StringBuffer sb)</code> Compares this string to the specified <code>StringBuffer</code> .
static <code>String</code>	<code>copyValueOf(char[] data)</code> Returns a <code>String</code> that represents the character sequence in the array specified.
static <code>String</code>	<code>copyValueOf(char[] data, int offset, int count)</code> Returns a <code>String</code> that represents the character sequence in the array specified.
boolean	<code>endsWith(String suffix)</code> Tests if this string ends with the specified suffix.
boolean	<code>equals(Object anObject)</code> Compares this string to the specified object.

Classes and Objects

- An **object**
 - is a single instance of a class
 - i.e. an object is created (instantiated) from a class.



Classes and Objects – Many Objects

- Many **objects** can be constructed from a single **class** definition.
- Each **object** must have a unique name within the program.

Ver 1.0

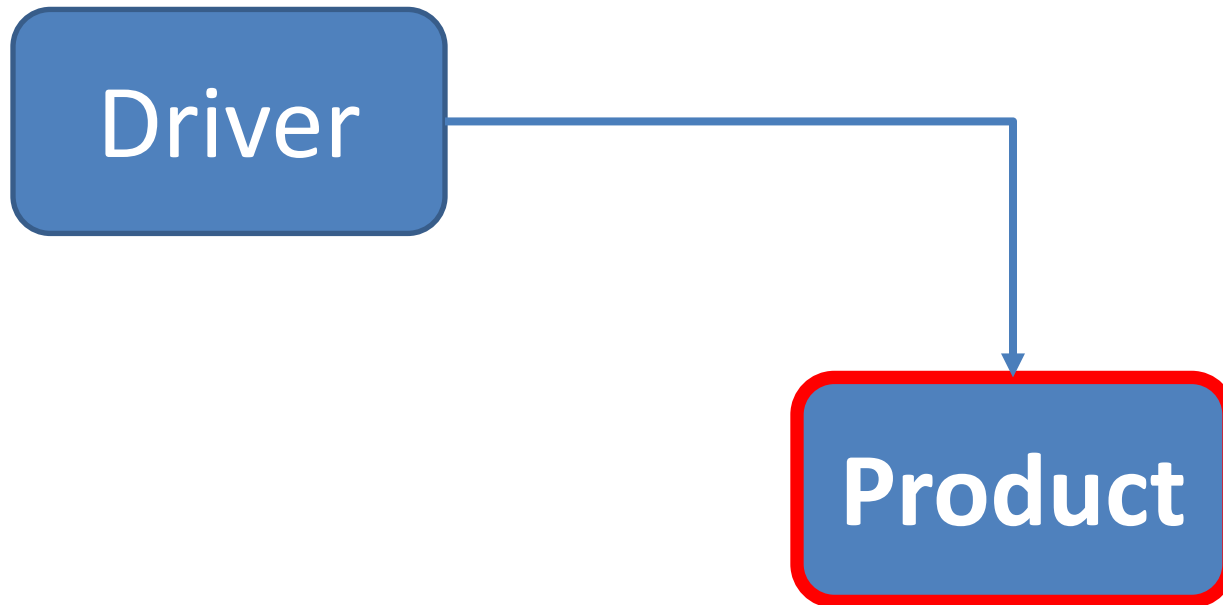
SHOP



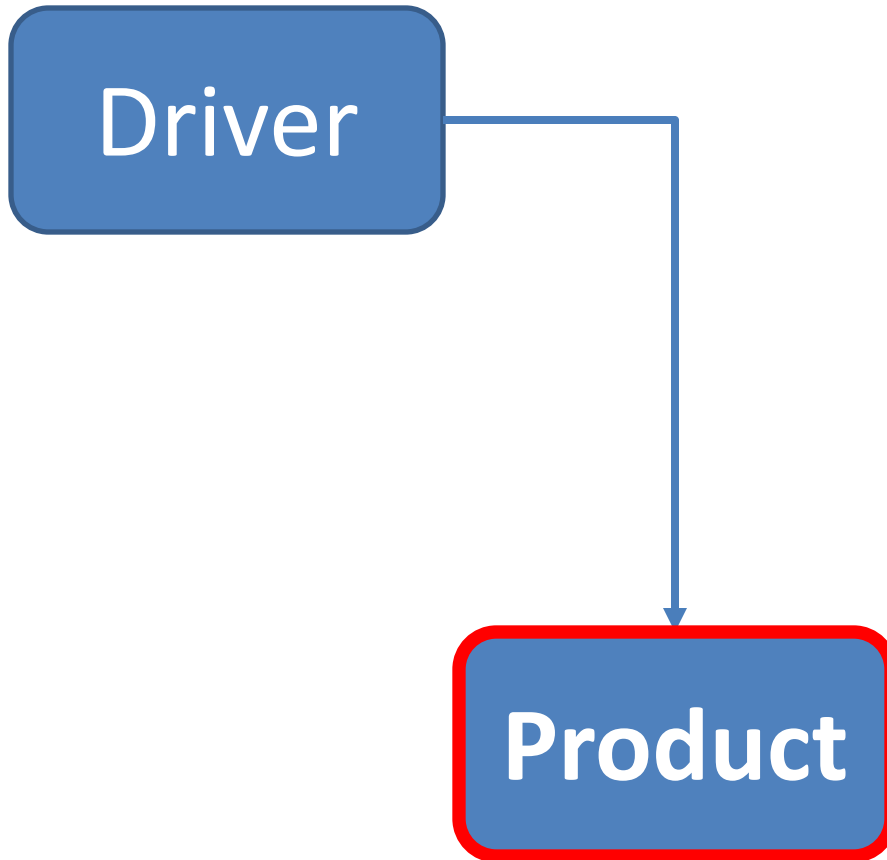
Shop V1.0 - Product



- We will recap object oriented concepts through the study of a new class called **Product**.



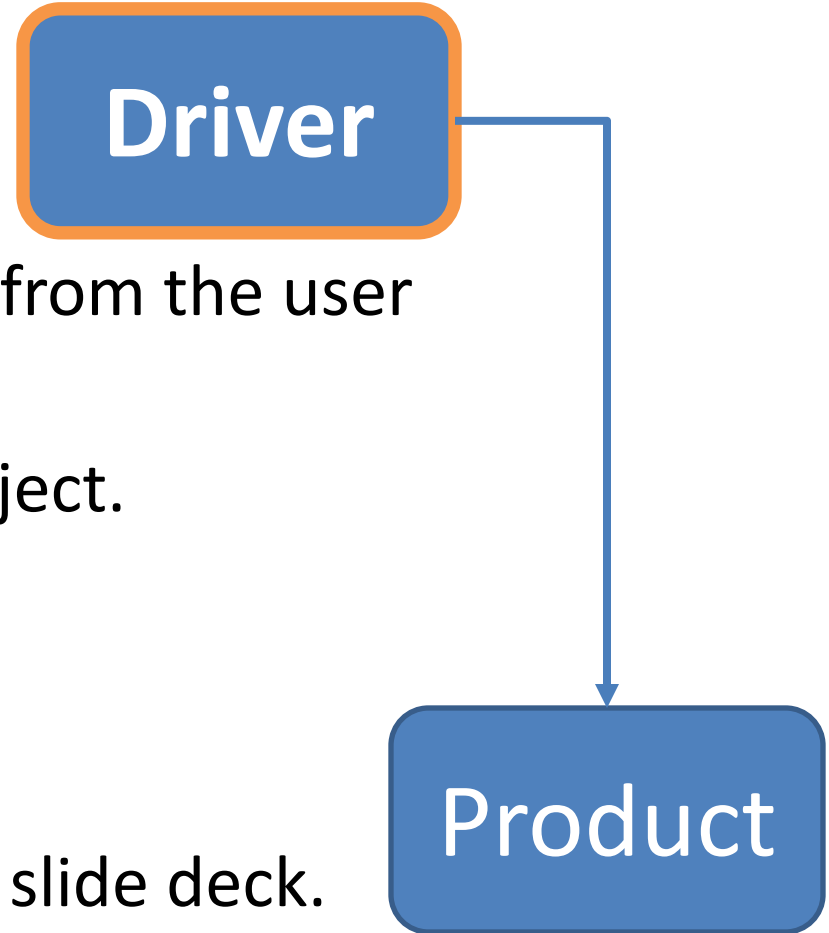
Shop V1.0 - Product



- The **Product** class stores **details** about a product
 - name
 - code
 - unit cost
 - in the current product line or not?

Shop V1.0 - Driver

- The **Driver** class
 - has the **main()** method.
 - **reads** the product details from the user (via the console)
 - **creates** a new Product object.
 - **prints** the product object (to the console)
- **Driver** is covered in the next slide deck.



A Product Class...



Object Type/ **Class** Name
i.e. Product

The **C** icon means it is a **Class**.

The open padlock means it is **public**.

```
✓ C 🔒 Product
  m 🔒 Product(String, int, double, boolean)
  m 🔒 getProductName(): String
  m 🔒 getUnitCost(): double
  m 🔒 getProductCode(): int
  m 🔒 isInCurrentProductLine(): boolean
  m 🔒 setProductCode(int): void
  m 🔒 setProductName(String): void
  m 🔒 setUnitCost(double): void
  m 🔒 setInCurrentProductLine(boolean): void
  m 🔒 toString(): String ↑Object
  f 🔒 productName: String
  f 🔒 productCode: int
  f 🔒 unitCost: double
  f 🔒 inCurrentProductLine: boolean
```

A Product Class...fields

field **type**

field **name**

The closed padlock means it is **private**.

The **f** icon means it is a **field**.

Fields
i.e. the **attributes / properties**
of the class

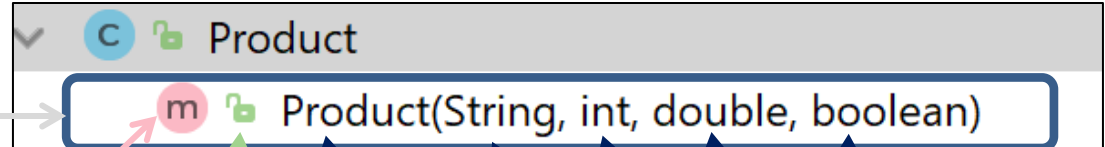
```
Product
  m Product(String, int, double, boolean)
  m getName(): String
  m getUnitCost(): double
  m getProductCode(): int
  m isInCurrentProductLine(): boolean
  m setProductCode(int): void
  m setName(String): void
  m setUnitCost(double): void
  m setInCurrentProductLine(boolean): void
  m toString(): String ↑Object
  f productName: String
  f productCode: int
  f unitCost: double
  f inCurrentProductLine: boolean
```

The screenshot shows the Product class with its methods and private fields. The fields are highlighted with an orange box. The annotations explain the field type, name, and the meaning of the 'f' icon and closed padlock.

A Product Class... constructor

Constructor

i.e. for building objects.



The **m** icon means it is a **method**.

The open padlock means it is **public**.

Constructors have same name as the class

Four **parameters**;
one for each field.

A Product Class... fields and constructor

```
public class Product {
```

```
    private String productName;  
    private int productCode;  
    private double unitCost;  
    private boolean inCurrentProductLine;
```

```
    public Product (String productName, int productCode,  
                   double unitCost, boolean inCurrentProductLine) {  
  
        this.productName = productName;  
        this.productCode = productCode;  
        this.unitCost = unitCost;  
        this.inCurrentProductLine = inCurrentProductLine;  
    }  
}
```

A Product Class... methods

Return type

Method name

The open padlock means it is **public**.

The **m** icon means it is a **method**.

Methods
i.e. the **behaviours** of the class

```
Product
├── Product(String, int, double, boolean)
├── getProductName(): String
├── getUnitCost(): double
├── getProductCode(): int
├── isInCurrentProductLine(): boolean
├── setProductCode(int): void
├── setProductName(String): void
├── setUnitCost(double): void
├── setInCurrentProductLine(boolean): void
├── toString(): String ↑Object
├── productName: String
├── productCode: int
├── unitCost: double
└── inCurrentProductLine: boolean
```

The screenshot shows the class structure of a `Product` class. The methods are listed with their return types and parameters. The `getProductName()` method is highlighted with a red box and underlined. Annotations explain the symbols used in the IDE: a blue circle with 'C' for class, a green 'm' icon for method, and an open padlock icon for public access. A red box highlights the entire methods section of the class.

A Product Class... **getters**

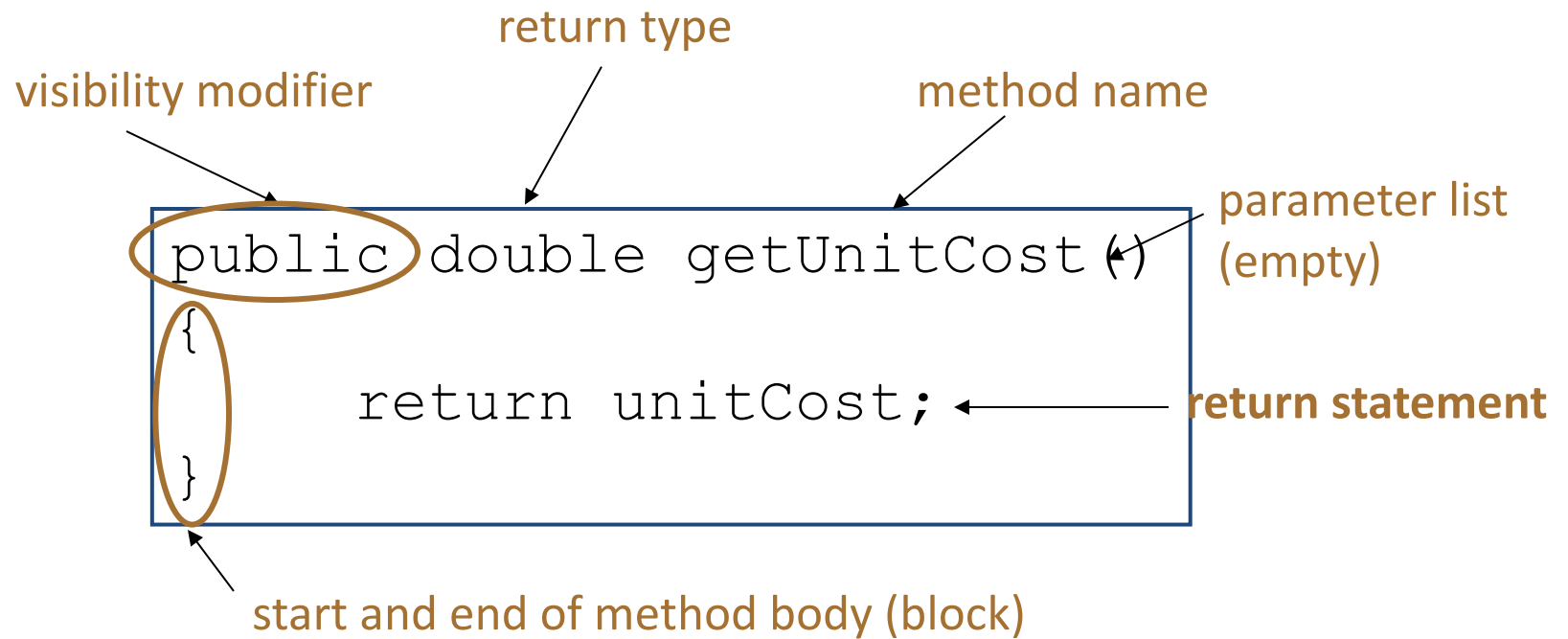
getters

```
Product
  m Product(String, int, double, boolean)
  m getProductName(): String
  m getUnitCost(): double
  m getProductCode(): int
  m isInCurrentProductLine(): boolean
  m setProductCode(int): void
  m setProductName(String): void
  m setUnitCost(double): void
  m setInCurrentProductLine(boolean): void
  m toString(): String ↑Object
  f productName: String
  f productCode: int
  f unitCost: double
  f inCurrentProductLine: boolean
```

Getters (Accessor Methods)

- **Accessor** methods
 - return information about the **state** of an object
 - i.e. the values stored in the fields.
- A **'getter'** method
 - is a specific type of **accessor** method and typically:
 - **contains a return statement**
(as the last executable statement in the method).
 - defines a **return type**.
 - **does NOT change the object state**.

Getters



A Product Class...getters

```
public String getProductName() {  
    return productName;  
}  
  
public double getUnitCost() {  
    return unitCost;  
}  
  
public int getProductCode() {  
    return productCode;  
}  
  
public boolean isInCurrentProductLine() {  
    return inCurrentProductLine;  
}
```

A Product Class...setters

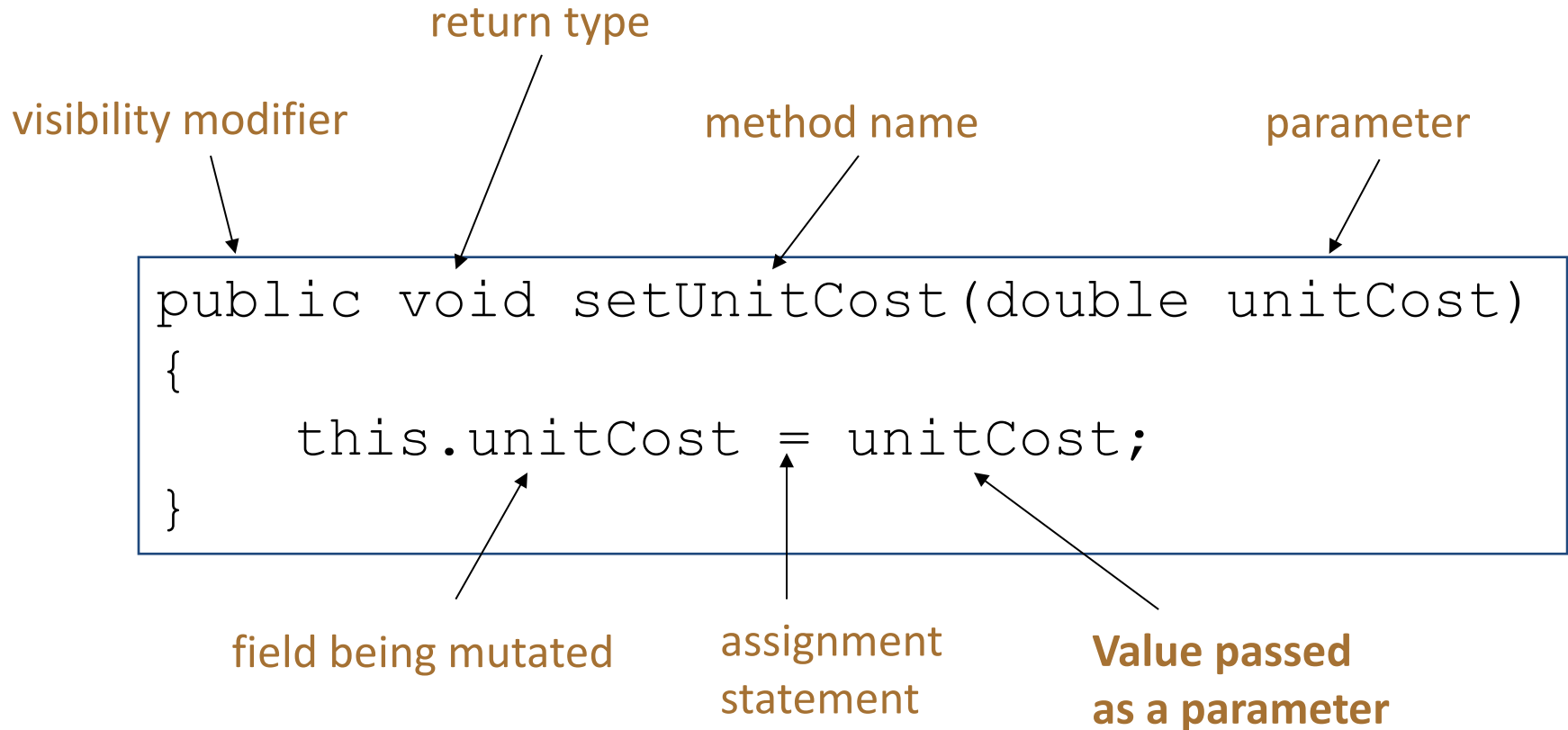
setters

```
Product
  m Product(String, int, double, boolean)
  m getProductName(): String
  m getUnitCost(): double
  m getProductCode(): int
  m isInCurrentProductLine(): boolean
  m setProductCode(int): void
  m setProductName(String): void
  m setUnitCost(double): void
  m setInCurrentProductLine(boolean): void
  m toString(): String ↑Object
  f productName: String
  f productCode: int
  f unitCost: double
  f inCurrentProductLine: boolean
```

Setters (Mutator methods)

- **Mutator** methods
 - change (i.e. mutate!) an object's state.
- A **'setter'** method
 - is a specific type of **mutator** method and typically:
 - contains an **assignment statement**
 - takes in a **parameter**
 - **changes the object state.**

Setters



A Product Class...setters

```
public void setProductCode(int productCode) {
    this.productCode = productCode;
}

public void setProductName(String productName) {
    this.productName = productName;
}

public void setUnitCost(double unitCost) {
    this.unitCost = unitCost;
}

public void setInCurrentProductLine(boolean inCurrentProductLine) {
    this.inCurrentProductLine = inCurrentProductLine;
}
```

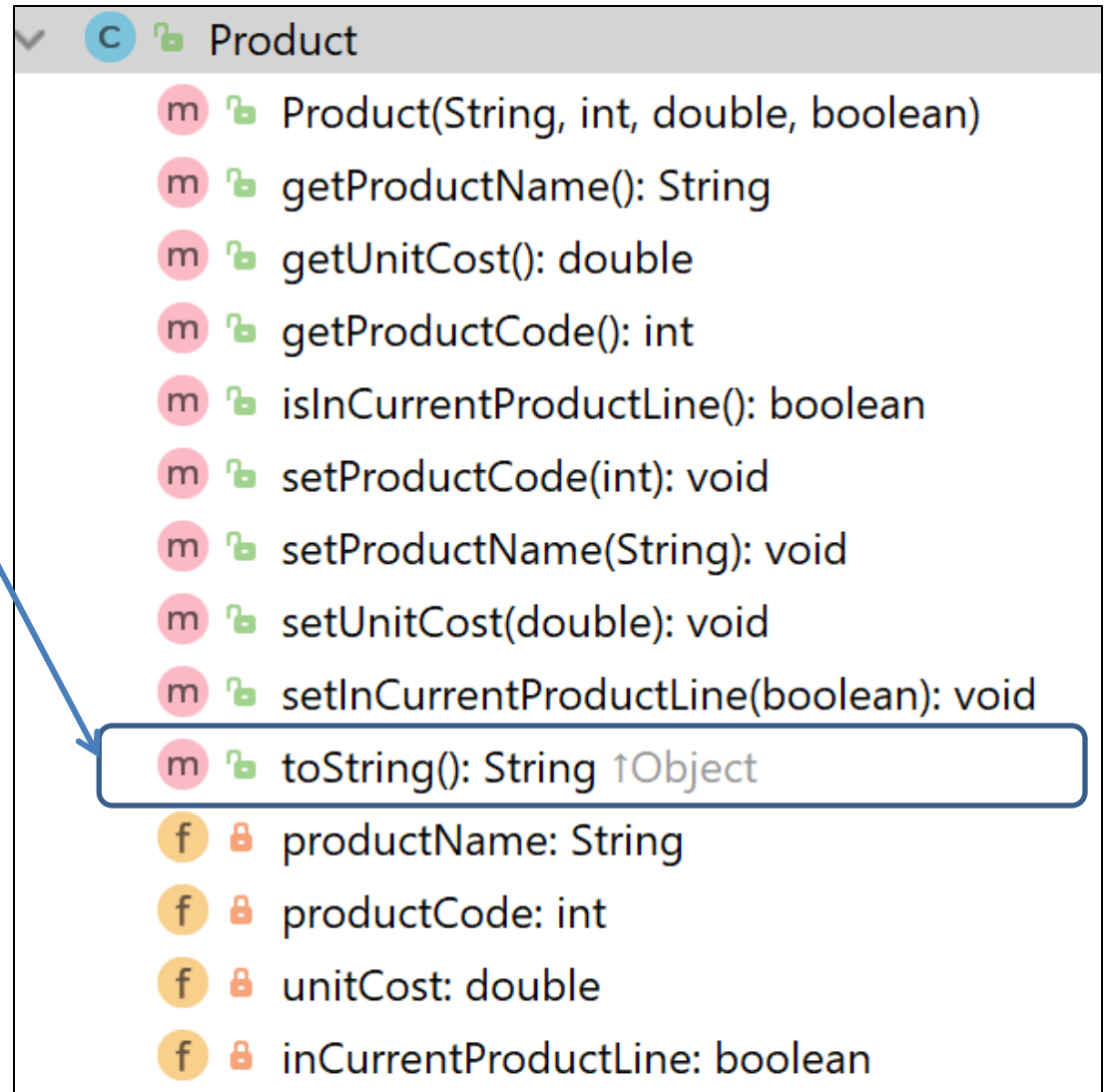
Getters/Setters

- For **each instance field** in a class, you are normally asked to write:
 - A **getter**
 - Return statement
 - A **setter**
 - Assignment statement

A Product Class...toString

toString():

Builds and returns a String containing a user friendly representation of the object state.



The screenshot shows the 'Product' class in an IDE. The class is expanded to show its methods and fields. The `toString(): String` method is highlighted with a blue box. A blue arrow points from the text box on the left to this method. The class also includes several other methods for getting and setting attributes, and five private fields.

```
Product  
  m Product(String, int, double, boolean)  
  m getProductName(): String  
  m getUnitCost(): double  
  m getProductCode(): int  
  m isInCurrentProductLine(): boolean  
  m setProductCode(int): void  
  m setProductName(String): void  
  m setUnitCost(double): void  
  m setIsInCurrentProductLine(boolean): void  
  m toString(): String ↑Object  
  f productName: String  
  f productCode: int  
  f unitCost: double  
  f isInCurrentProductLine: boolean
```


A Product Class...

```
public String toString()
{
    return "Product description: " + productName
        + ", product code: " + productCode
        + ", unit cost: " + unitCost
        + ", currently in product line: " + inCurrentProductLine;
}
```

Sample Console Output if we printed a Product Object:

Product description: 24 Inch TV, product code: 23432, unit cost: 399.99, currently in product line: true

toString()

- This is a useful method and you will write a **toString()** method for most of your classes.
- **When you print an object, Java automatically calls the toString() method** 
e.g.

```
Product product = new Product();  
  
//both of these lines of code do the same thing  
System.out.println(product);  
System.out.println(product.toString());
```

Encapsulation in Java – steps 1-3

Encapsulation Step	Approach in Java
1. Wrap the data (fields) and code acting on the data (methods) together as single unit.	<pre>public class ClassName { Fields Constructors Methods }</pre>
2. Hide the fields from other classes.	Declare the fields of a class as <u>private</u>.
3. Access the fields only through the methods of their current class.	Provide <u>public</u> setter and getter methods to modify and view the fields values.

A Product Class... An Encapsulated Class

1. Product class **wraps** the data (fields) and code acting on the data (methods) together as **single unit**.

2. Fields are **hidden** from other classes.

3. **Access** the fields only through the methods of Product (e.g. **getter** and **setter** methods).

The screenshot shows the source code of a Java class named `Product`. The class contains a constructor and several methods. The fields are private and accessed via getters and setters. Red boxes and arrows highlight the encapsulation features:

- A red box highlights the four getter methods: `getProductName(): String`, `getUnitCost(): double`, `getProductCode(): int`, and `isInCurrentProductLine(): boolean`. A red arrow points from the text "single unit" in the first box to this group.
- A red box highlights the four setter methods: `setProductCode(int): void`, `setProductName(String): void`, `setUnitCost(double): void`, and `setInCurrentProductLine(boolean): void`. A red arrow points from the text "getter and setter methods" in the third box to this group.
- A red box highlights the four private fields: `productName: String`, `productCode: int`, `unitCost: double`, and `inCurrentProductLine: boolean`. A red arrow points from the text "Fields are hidden" in the second box to this group. A red circle is drawn around the lock icon next to the first field, `productName`.

Using the Product Class

1

```
private Product product;
```

Declaring an object
product, of type
Product.

product

null

Using the Product Class

1
`private Product product;`

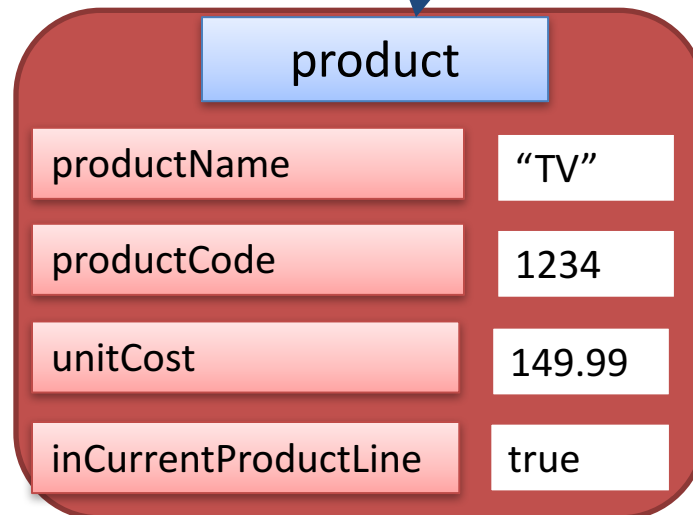
Declaring an object **product**, of type **Product**.

product



2
`product = new Product("TV", 1234, 149.99, true);`

Calls the **Product** constructor to build the **product** object in memory.



Multiple Product Objects

```
private Product product = new Product("TV", 1234, 149.99, true);
```

product



product

productName

"TV"

productCode

1234

unitCost

149.99

inCurrentProductLine

true

Multiple Product Objects

```
private Product product = new Product("TV", 1234, 149.99, true);
```

```
private Product phone = new Product("iPhone 3", 1001, 349.99, false);
```

product



product

productName

"TV"

productCode

1234

unitCost

149.99

inCurrentProductLine

true

phone



phone

productName

"iPhone8"

productCode

1001

unitCost

799.99

inCurrentProductLine

false

Questions?

