Understanding class definitions 2

Exploring source code

- Produced by: Ms. Mairead Meagher, Ms. Siobhán Roche.

Ollscoil Teicneolaíochta an Oirdheiscirt
South East Technological University

Department of Computing and Mathematics
http://www.setu.ie/

Upcoming

- Methods, including:
  - *accessor* (*getter*) methods
  - *mutator* (*setter*) methods;
- String formatting;
- Conditional statements;
- Local variables. (next slidedeck)

Methods

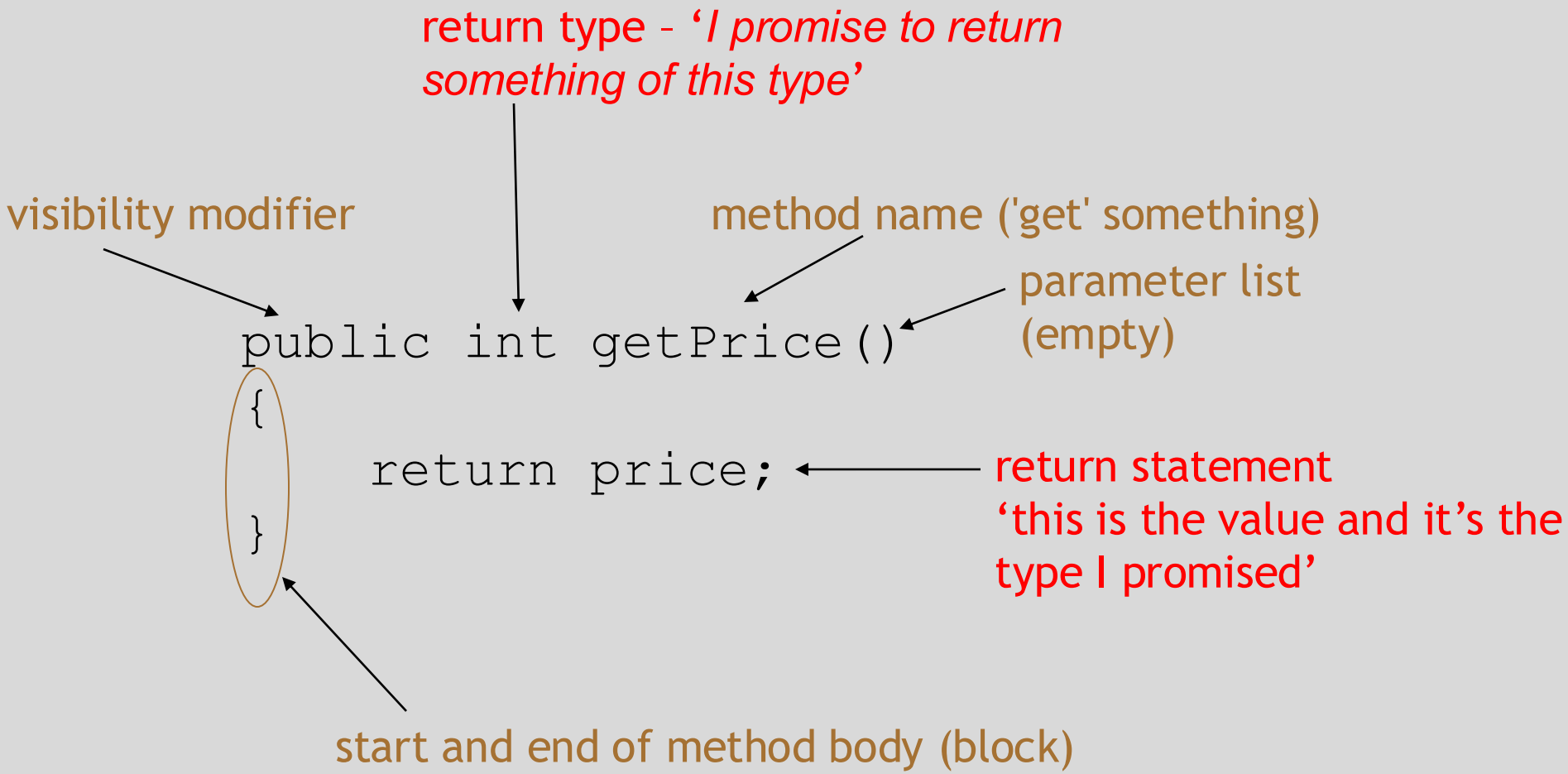
- Methods implement the *behaviour* of objects.
- Methods have a consistent structure comprised of a *header* and a *body*.
- *Accessor methods* provide information about an object.
- *Mutator methods* change the state of an object.
- Other sorts of methods accomplish a variety of tasks.

Method structure

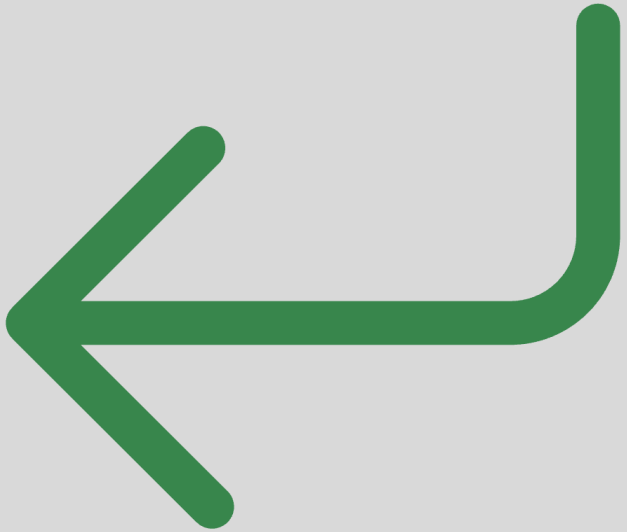
- Accessor methods often occur in the form of *getter* methods.
- The header:
  - **`public int getPrice()`**
- The header tells us:
  - the *visibility* to objects of other classes (public);
  - whether the method *returns a result* (an integer);
  - the *name* of the method *(*getPrice*);*
  - whether the method takes *parameters* (none, in this case).

- The body encloses the method's *statements*.

Getter methods



Accessor methods



- An accessor method always has a return type that is not void.
- An accessor method returns a value (result) of the type given in the header.
- The method will contain a return statement to return the value.
- NB: Returning is not printing!
- It presents a value to the calling method.

Test – syntax errors

```
public class CokeMachine
{
    private price;

    public CokeMachine()
    {
       price = 300
    }

    public int getPrice
    {
       return Price;
    }
```

- What is wrong here? There are five errors.

Test

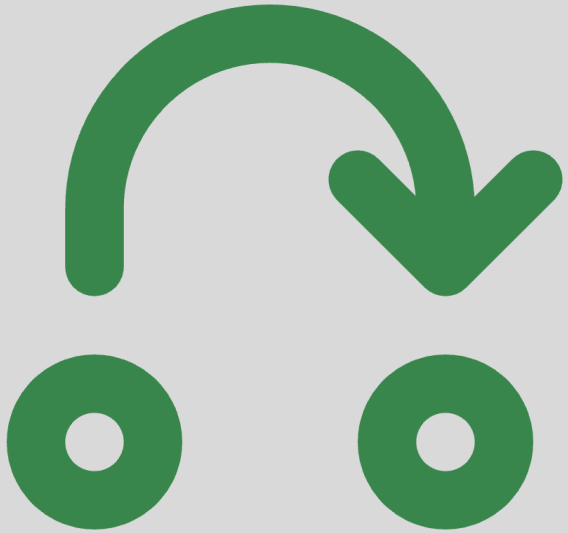
```
public class CokeMachine
{
        int
    private price;

    public CokeMachine()
    {
        price = 300 ;
    }

    public int getPrice ()
    {
        return Price;
    }
}
```

- What is wrong here? There are five errors.

Mutator methods



- Have a similar method structure: header and body.
- Used to *mutate* (i.e., change) an object's state.
- Achieved through changing the value of one or more fields.
  - They typically contain one or more assignment statements.
  - Often receive parameters.
- The most basic mutator methods are 'setter' methods.

setter methods

- Fields often have dedicated **`set`** mutator methods.

- These have a simple, distinctive form:
  - **`void`** return type
  - method name related to the field name
  - single formal parameter, with the same type as the type of the field
  - a single assignment statement

A typical **`set`** method

```
public void setDiscount(int amount)
{
    discount = amount;
}
```

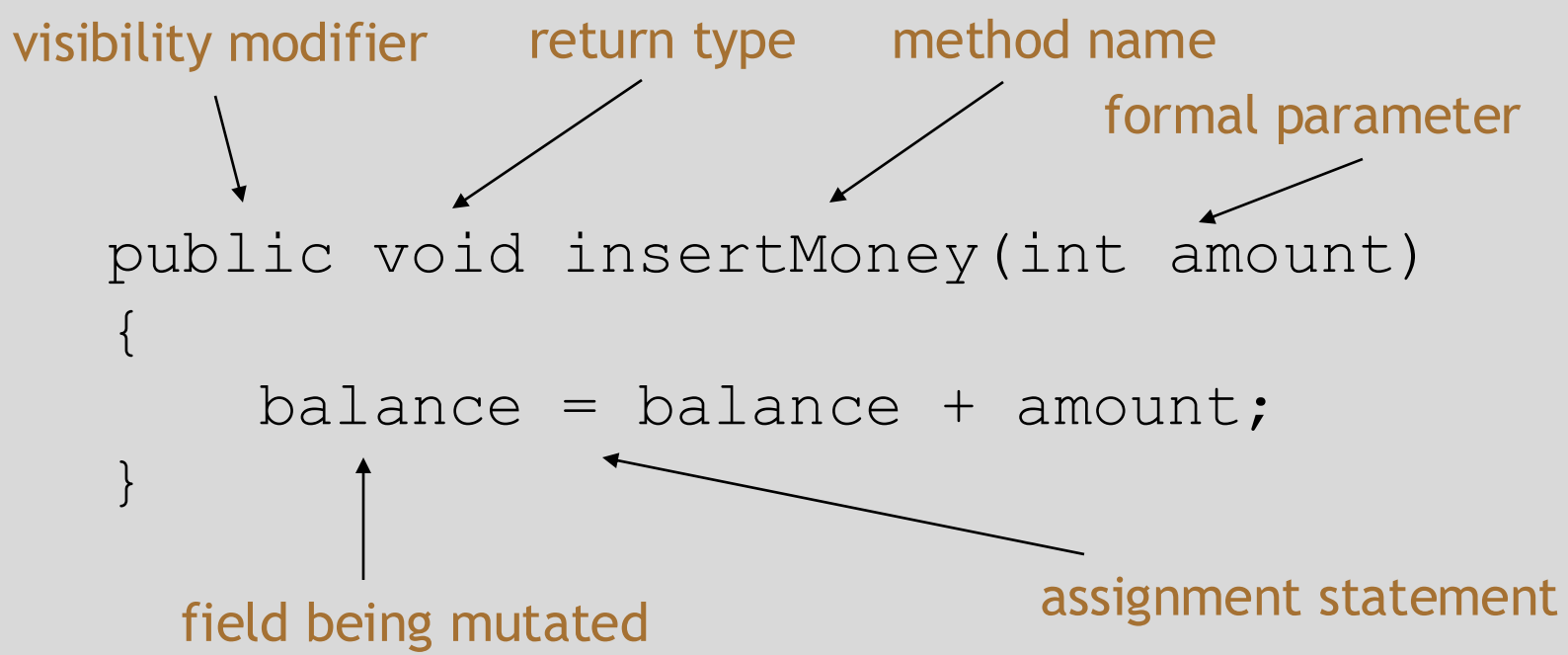
We can easily infer that **`discount`** is a field of type **`int`**, i.e:

```
private int discount;
```

Protective mutators

- A set method does not have to always assign unconditionally to the field.
- The parameter may be checked for validity and rejected if inappropriate.
  - e.g. only update field if new value is >=10
- Mutators thereby protect fields.
- Mutators support *encapsulation*.

Another mutator method



This method *changes* the value of balance; hence the object's state is *mutated*.

The `this` keyword

- Up to now we have come up with a different name for the parameter and the field.
- However, sometimes there is one name that perfectly describes the use of a variable—it fits so well that we do not want to invent a different name for it.

- **`Enter the this. construct`**

The `this` keyword

- Used to distinguish parameters and fields of the same name.
  E.g.:

```
public TicketMachine(int price)
{
    this.price = price;
    balance = 0;
    total = 0;
}
```

The `this` keyword

- Used to distinguish parameters and fields of the same name. E.g.:

```
public class TicketMachine
{
    private int price;
    private int balance;
    private int total;


public TicketMachine(int price)
{
    this.price = price;
    balance = 0;
    total = 0;
}
}
```

The `this` keyword

```
this.price = price;
```

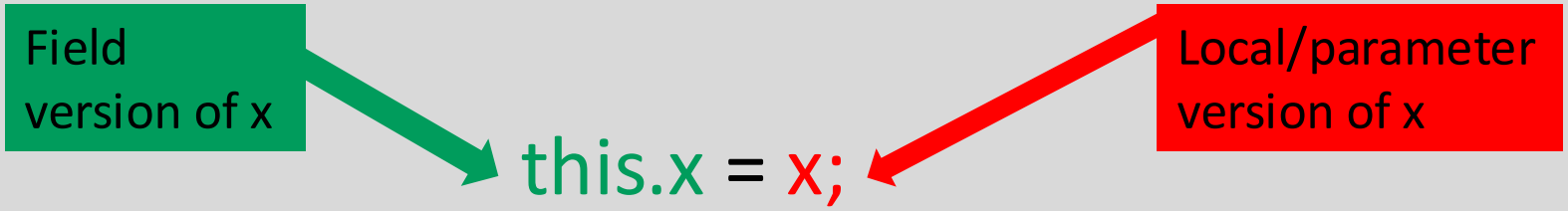
This statement has the following effect:

field named price = *parameter named* price;

The **this** keyword

- The **this** keyboard can **always** be used when referencing field variables.

- In practice, we use it when there is a parameter with the same name as a field.



Printing from methods

```
public void printTicket()
{
    // Simulate the printing of a ticket.
    System.out.println("##################");
    System.out.println("# The BlueJ Line");
    System.out.println("# Ticket");
    System.out.println("# " + price + " cents.");
    System.out.println("##################");
    System.out.println();

    // Update the total collected with the balance.
    total = total + balance;
    // Clear the balance.
    balance = 0;
}
```

String concatenation

- 4 + 5

  9
- "wind" + "ow"

  "window"
- "Result: " + 6

  "Result: 6"
- "# " + price + " cents"

  "# 500 cents"

Quiz

- System.out.println(5 + 6 + "hello");

- System.out.println("hello" + 5 + 6);

Quiz

- System.out.println(5 + 6 + "hello");
- System.out.println("hello" + 5 + 6);

11hello

hello56

Formatted printing

- Concatenation can be used to create output in a desired format.
- An alternative is to use **`printf`**.
- The first argument gives the overall structure.
- Format specifiers represent 'holes' to be filled in by the other arguments.

Formatted printing

- Concatenation:
```
System.out.println("# " + price + "
cents.");
```
- Using **`printf`**:
```
System.out.printf("# %d cents.%n",
price);
```
- **`%d`** means insert the next argument as an integer value at this point.
- **`%n`** means 'end the line' (i.e., 'insert' a newline) at this point.

Method summary

- Methods implement all object behavior.
- A method has a name and a return type.
  - The return-type may be **`void`**.
  - A non-**`void`** return type means the method will return a value to its caller.
- A method might take parameters.
  - Parameters bring values in from outside for the method to use.

Reflecting on the ticket machines

- Their behavior is inadequate in several ways:
  - No checks on the amounts entered.
  - No refunds.
  - No checks for a sensible initialization.
- How can we do better?
  - We need the ability to choose between different courses of action.

Questions?