Programming Foundations
FIT9131

Defining a class

Week 2

Lecture outline

- review of OO concepts from last week
- defining a class
  - fields
  - assignments
  - expressions
  - constructors
  - methods and method signatures
    - parameter passing, return values
    - "accessor" and "mutator" methods
- data types
- displaying information

Review : Objects and Classes

- *objects*
  - represent 'things' from the real world, or from some problem domain (example: "the *customers* for a local library")
- *classes*
  - represent all objects of a particular kind (example: '*Cat*').

- multiple *instances* (ie. *objects*) can be created from a single *class*. In a way, a class is like a template used to produce objects.

Review : Attributes, behaviour, state, identity

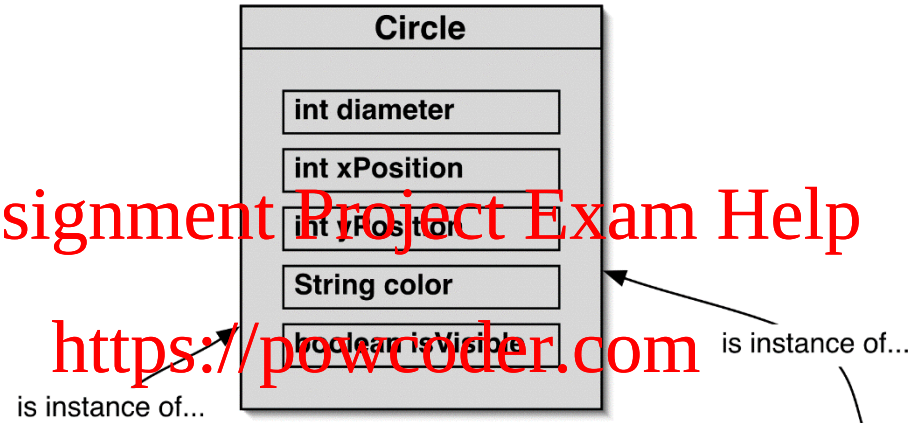
- An object has *attributes* and *behaviour*.
- The class defines the *attributes* of an object, but each object stores its own set of values for its attributes (the *state* of the object). (eg. an object of the class Cat could have a 'colour' attribute, and the colour of a particular cat may be "brown"). These values define the "*state*" of an object.
- The class also defines the *behaviour* of the object. (For example: a Cat object can *climb, jump, etc*)

- Each object has an "*identity*", which makes it unique from any other object. In other words, *all objects are different.*

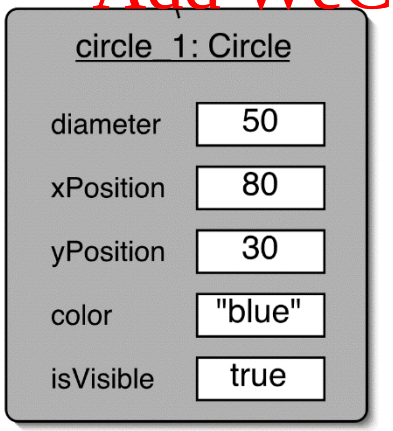
Example : a Circle *Class* & two Circle *Objects*

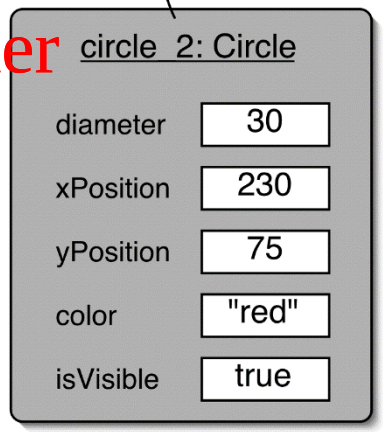
Circle *Class*

An example of how we can represent Classes & Objects using diagrams



Circle *Object* #1

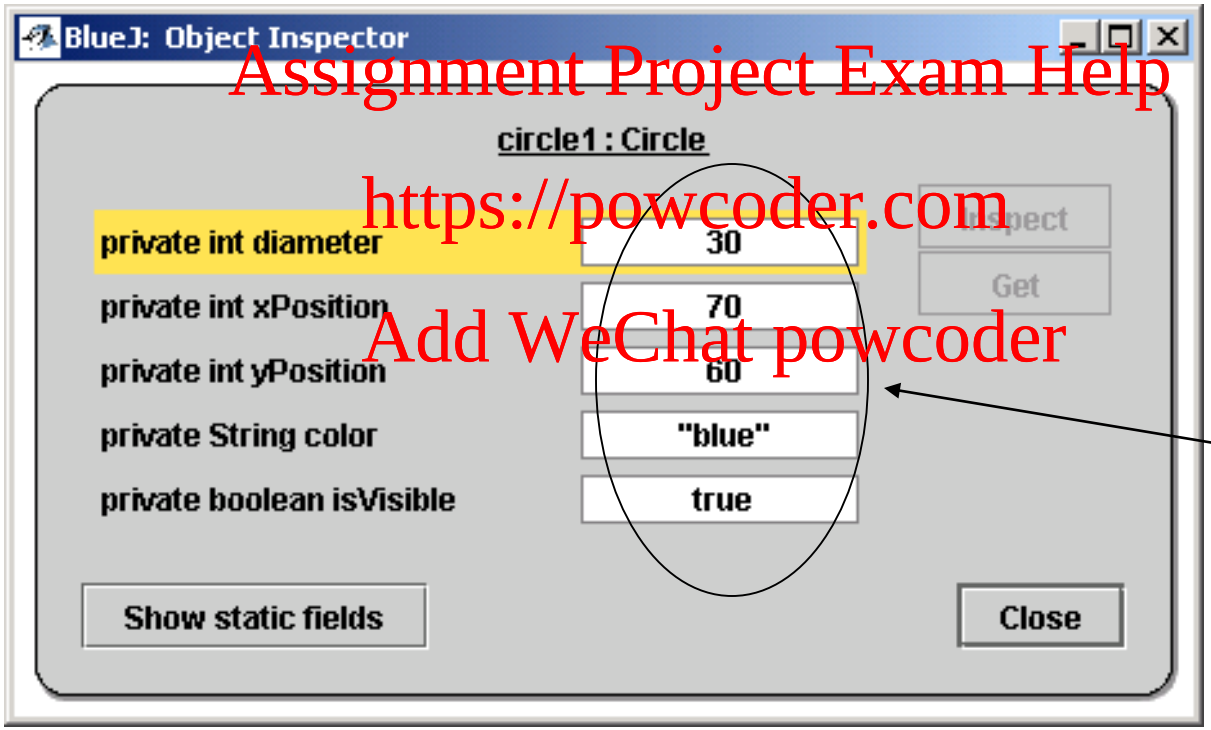




Circle *Object* #2

The State of a Circle object

The "*State*" of an object refers to the ***values of its attribute*** at a particular moment in time.
BlueJ allows us to examine an object's state with its ***Inspector***.



The "state" of this Circle object

Defining a class in Java

Each class is defined by its source code (Java code).

Basic elements of a Java class definition:

- *fields* (sometimes known as *instance variables*)
- *constructors* (special methods to *create* objects)
- *methods* (these define the objects' *behaviour*)

We will look at an example of a class definition using the *naïve-ticket-machine* project from the BlueJ *Projects* directory (chapter02).

- these sample projects are available for download from the BlueJ website (or the FIT9131 website)

Java Classes and Programs

- A typical Java program is made up of many Java classes. Generally, one of these classes acts as the "*command centre*" to control all the other classes. During the lifetime of the program, objects are created from the various classes, and the objects interact with each other (to make the program perform some specific tasks).

- The *naive-ticket-machine* sample program consists of only one Java class ("`TicketMachine.java`"). BlueJ stored this in a "project" named "`naive-ticket-machine`".

TicketMachine – logic

- The ticket machine works by user "inserting money" into it, and then requesting a ticket to be printed.

- This is a *"naive"* machine because:
  - it does not check the money entered to see if it is enough for the ticket
  - it does not give a refund if too much money is entered
  - it does not check to ensure that the customer enters a sensible amount of money
  - It does not check that the ticket price passed to its constructor is sensible

TicketMachine – an external view

Exploring the behavior of a ticket machine from the *naive-ticket-machine* project.

- This machine supplies tickets of one fixed price.
- Other behaviour …?

Some questions we might ask:

- How is the ticket price determined?
- How is 'money' entered into a machine?
- How does a machine keep track of the money that is entered?
- How is the ticket 'printed'?

TicketMachine – an internal view

Interacting with an object (by *calling its methods*) gives us clues about its behaviour.

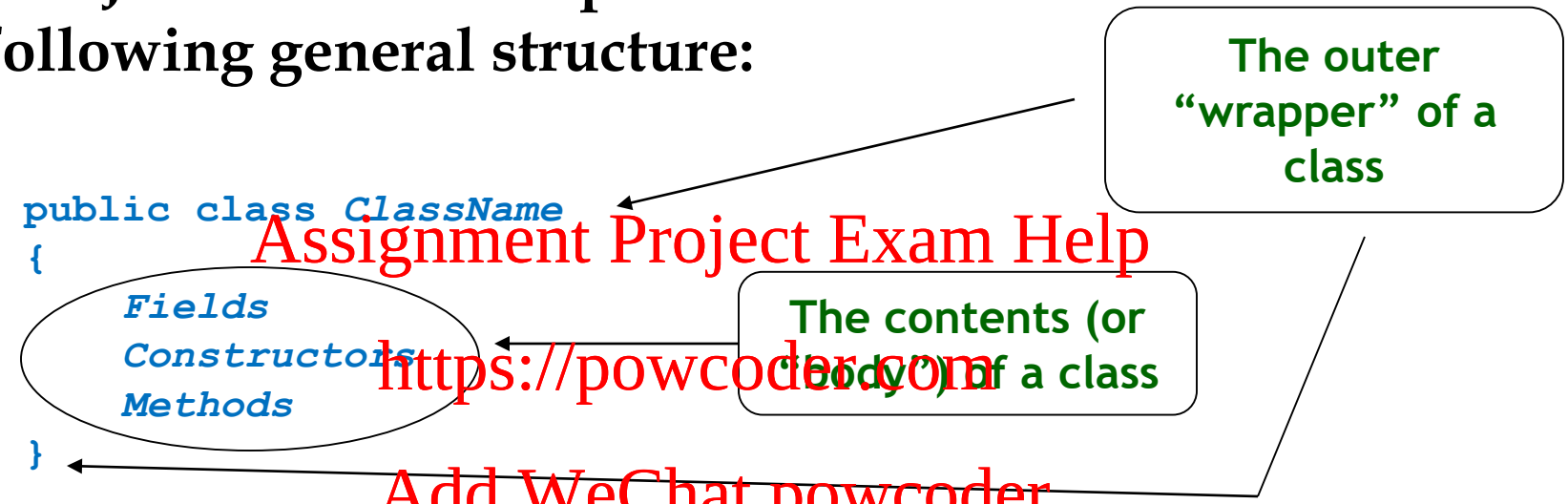
Looking inside (by examining the Java code) allows us to determine how that behaviour is implemented.

All Java classes have a similar-looking internal view (or code structure).

BlueJ provides an interface for us to perform the above tasks easily.

Basic class structure

The Java code that implements a class has the following general structure:



The Java code for the TicketMachine example:

```
public class TicketMachine
{
    Inner part of the class omitted here for brevity…
}
```

Terminologies

We have learnt some common OO terminology.

Next we will also look at some common programming terms.

You will need to understand all these in order to understand important programming concepts.

Language Syntax

- In programming, the word "*Syntax*" is used to describe the *structure of statements* in a computer language (i.e. how the statements are to be written).

- For programs to be able to be translated/executed by computers, the statements need to be written in a well-defined and well-structured manner.

- We will start by learning the Java language syntax. Most high-level languages have very similar syntax.

Statement

A Java *Statement* is an action, or series of actions, ending with a semicolon ';' .

Syntax :
```
JavaStatement;
```

Examples :
```
price = 0;
sum = number1 + number2;
```

Variables (introduction)

Variables are "containers" which are used by programs to store values. Most modern programming languages allow programmers to specify the *types* (eg. integers, strings, etc.) of the variables.

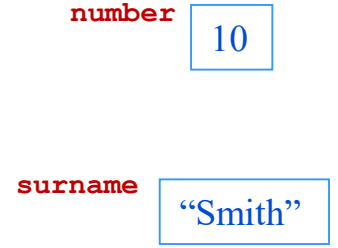
Variables typically need to be the "*declared*" before they can be used. What this usually means is that programmers need to explicitly specify what is the type of a variable, so that the computer can allocate sufficient memory space to store it.

Syntax : `type variableName = value;`

Examples :

```
int    number  =  10;
String surname =  "Smith";
```

Variable type — variable name — variable value



Fields

Fields store values (ie. the *attributes*) for an object. (Note that they are also known as *instance variables* but we will use the term *field*). We can treat fields as special variables.

Use the *Inspect* option in BlueJ to view an object's fields.

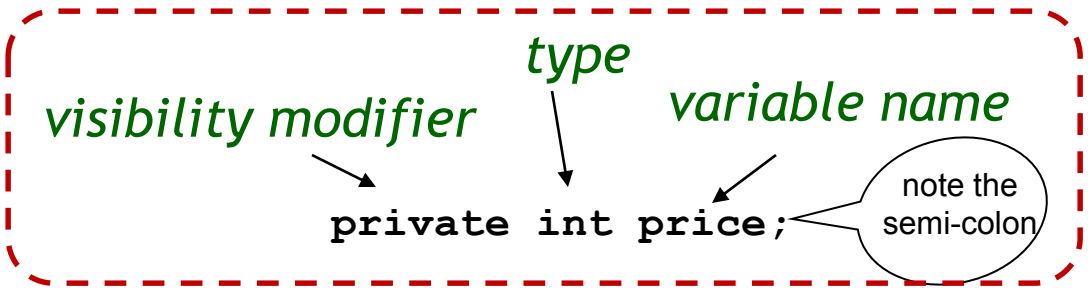
Fields define the *state* of an object.


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

    Constructor and methods omitted...
}
```

3 fields declared here

Syntax for declaring fields



Assignment Statements

Values are stored in fields (and other variables) via *assignment statements*.

In Java, the assignment operator is `=` It is read as "becomes", "takes on the value of ", or "is assigned".

Syntax : `variable = expression;`

Example : `price = ticketCost;`

Eg:


```
price = 50;
```

puts the value 50 into an existing variable named `price`.

Expressions

An expression returns a value. The value *replaces the expression* in the code. We *evaluate* the expression to get its value. Some examples:

```
ticketCost                  (a single-value expression)
balance + amount            (an arithmetic expression)
```


This "return value" can be assigned to a field (or another variable), or passed to a method as an argument, or used in another expression. Eg:

```
price = ticketCost;

balance = balance + amount;

return balance;
```

expressions

Methods

The behaviour of objects is implemented by defining *methods*.

Methods have two main parts: a *header* and a *body*. An optional *comment* describing what the method does should also be included.


```
/**
 * Return the price of a ticket
 */
public int getPrice()
{
  return price;
}
```

comment

header

body

Method signature

The header specifies who can access this method, what type of data it returns, its name, and what other information it needs. This information is necessary in order to be able to use the method.

The method name and the following parameter list is called the *signature* of the method.

method signatures

```
public int getPrice()
```



```
public void insertMoney(int amount)
```

The body is the set of instructions that tells the computer how to do the job that this method is designed to do.

Constructors

A class can create an instance of itself (i.e. creating an object). When it does so, a special method known as a *constructor* is called.

A constructor has the same name as the class.

A constructor's task is to assign initial values to the attributes of the new object that is being created.

They may also receive external parameter values to initialise some or all of the attributes.

Eg.

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

← Constructor for the **TicketMachine** class

Using Constructors

When a new object is created, it must be in a sensible state (ie. its attributes must have some sensible initial values).

We must write code in a class's constructor(s) to ensure that this condition is met.

A "*Default Constructor*" is a constructor which does not have any parameters. Typically, we first create a default constructor for a class, and then add other constructors as necessary. If no default constructor is specified for a class, Java automatically creates one.

Passing data via parameters

Parameters are defined in the *header* of the constructor (or header of a method).

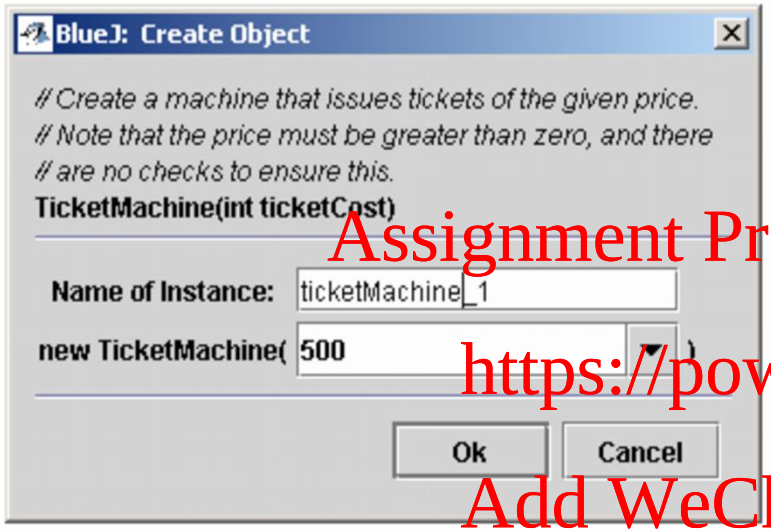
The following constructor has a single parameter of type `int` (integer) :

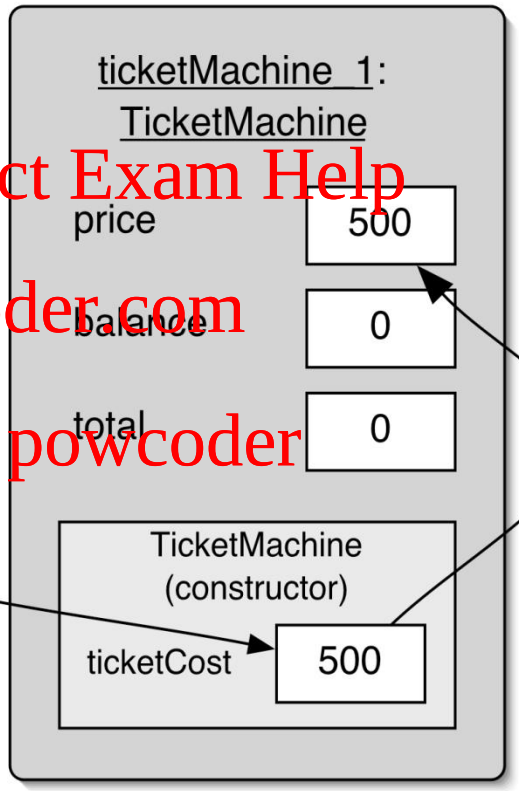
```
public TicketMachine(int ticketCost)
```


The parameter names inside a constructor or method are referred to as *formal parameters*, and parameter values outside (ie. when the method is actually called) are referred to as *actual arguments*.

Passing data via parameters

In BlueJ, the value (ie. actual argument) is entered via a dialogue box





(A)

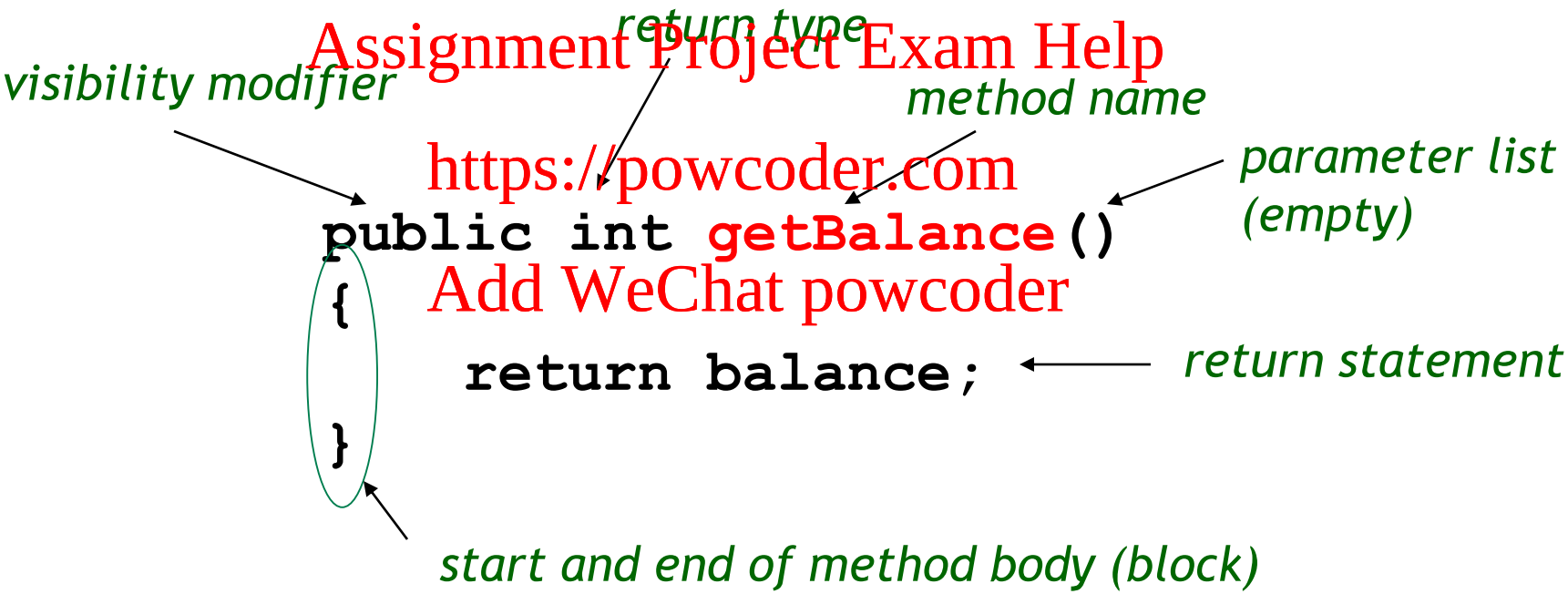
(B)

the constructor gets 500 as the parameter here

in this example, **ticketCost** is a ***formal parameter*** and **500** is an ***actual argument***. When a method is "called", its formal parameter is replaced by the actual arguments.

Accessor ("get") methods

Accessor methods are used to provide information about an object. They are also often called "*get* " methods.



The `return` statement

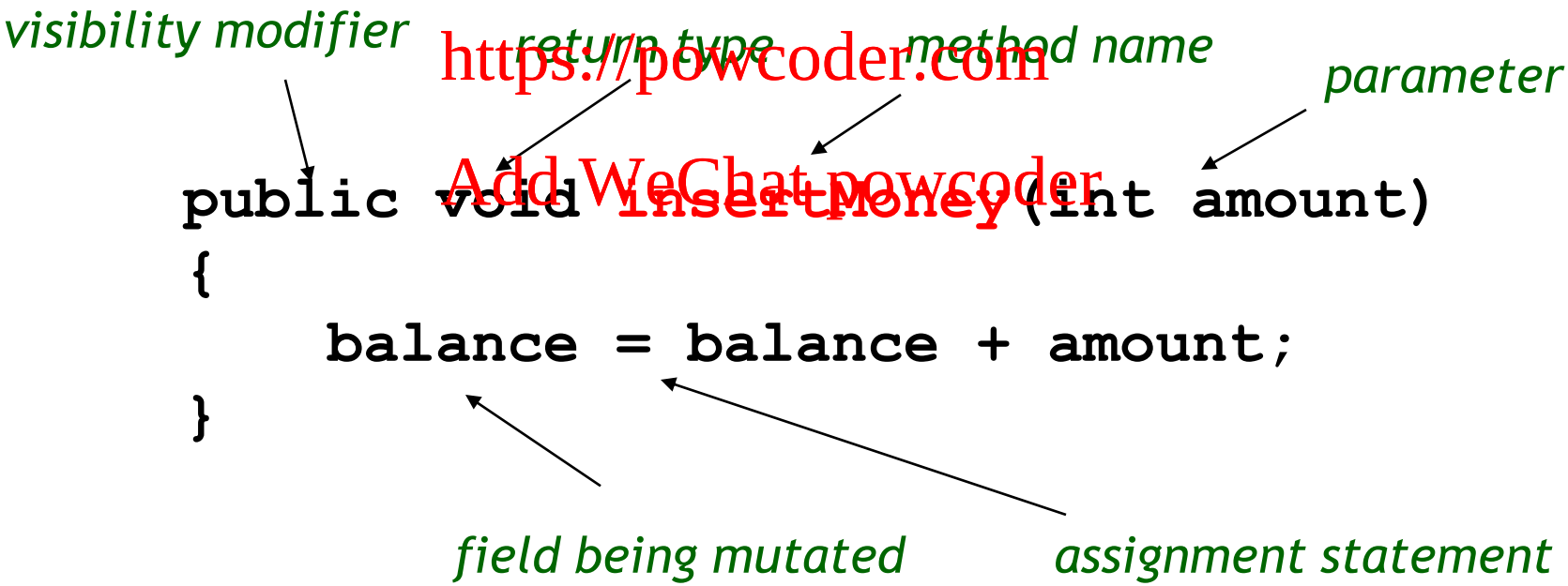
The `return` statement in a method does two things:

- it sends back a value to the calling method
- it causes the execution of its own method to be terminated immediately

This means that, if a method has a `return` statement, it should generally be the *last* statement in the method. If there is anything after the `return` statement, it will never be executed.

Mutator ("set") methods

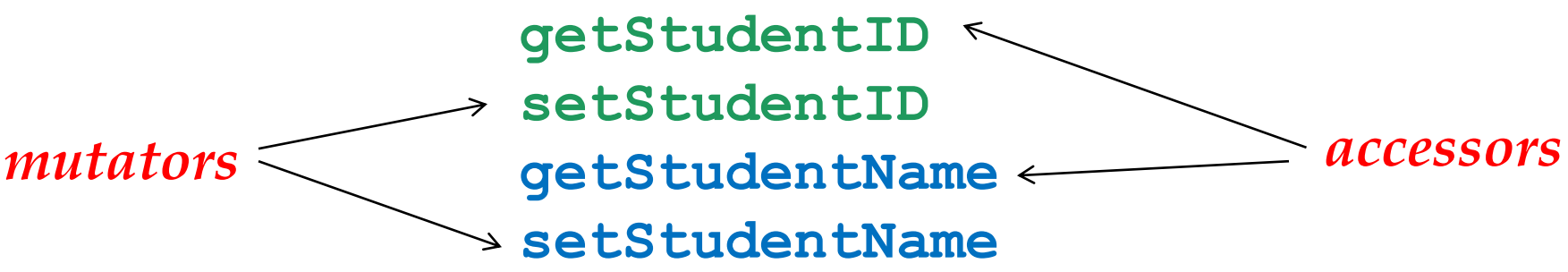
Mutator **methods are used to change the state of an object, by changing the value of one or more fields. Mutator methods typically receive parameters and contain assignment statements. Mutator methods are also called "*set* " methods.**



Attributes & Accessors/Mutators

In general, when you declare an attribute for a class, you should also declare a pair of *accessor* & *mutator* methods for it.

For example, if your class has 2 attributes named "`studentID`" and "`studentName`", then you should have at least these 4 methods:



Variables (re-visited)

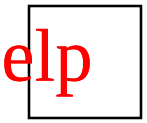
- *Fields* and *parameters* are special types of *variables*.
- A variable is a storage location in the computer's memory that is able to store a value – one piece of data.
- These values often change during the running of a program – hence the name *variable*.
- To define a variable we need to specify its *name* and the *type* of data that it will hold.
- We usually also specify the *value* that the variable holds.

Example : Creating a variable

We create (or *define*) a variable by declaring its *type* and its *name*:


```
int age;
```

=====>



age


```
String name;
```


=====>

name

A variable must be defined *before* it can be used.

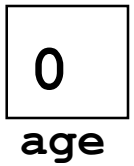
Initialising a variable

If a variable is not initialised, it may have a value of whatever happens to be in the memory location allocated to it. This is a bad programming practice.

Our coding standards specify that you must always allocate an *initial value* to a variable when you define it. This is called *initialising* the variable.

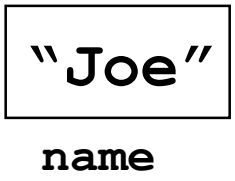

```
int age = 0;
```

=====>




```
String name = "Joe";
```

=====>



Data types in Java

When we create a variable in Java, we need to specify the type of the value we want to put in there.

The type of a piece of data tells Java what can be done with it, and how much memory needs to be put aside for it.

There are many different types of data. Some types are defined by Java – others may be defined by the programmer.

There are different operations that can be performed on data, depending on its type.

Primitive types versus Object types

Primitive types are predefined in the language. (e.g. `int`, `float`, `char`, `boolean`)

Primitive values are stored directly in a variable.

Object types are defined by classes. These are pre-defined (e.g. `String`) or we can write them ourselves.

Objects are not stored directly in a variable, instead, a *reference* to the object is stored. More about this in a later lecture …

We will now look at some of the primitive types …

Type int

An **int** (integer) is a whole number (e.g. 21). You can do arithmetic with an **int**.

```
int age = 21;      =====>
```

Arithmetic operators :

```
+   addition
-   subtraction
*   multiplication
/   division
%   modulus   (or remainder)
```

Arithmetic operators

Given the following definition for the variable age:

```
int age = 15;
```


What will these expressions evaluate to?

```
age + 3
2 * age – 4
2 * (age – 4)
```

Arithmetic operators

Given the following definition for the variable age:

```
int age = 15;
```


What will these expressions evaluate to?

```
age + 3          ➔ 18
2 * age – 4      ➔ 26
2 * (age – 4)    ➔ 22
```

Integer division and modulus

The operator / performs an integer division. The result is an integer. Any remainder will be discarded.

If you want to know what the remainder is, the operator % (called *modulus*) will give it to you.


```
int age = 15;

age / 2
age % 10
age % 20
```

Integer division and modulus

The operator / performs an integer division. The result is an integer. **Any remainder will be discarded**.

If you want to know what the remainder is, the operator % (called *modulus*) will give it to you.


```
int age = 15;

age / 2          ➔ 7
age % 10         ➔ 5
age % 20         ➔ 15
```

Whole number types

The `int` type (and other whole number types) are stored in a binary representation with one bit for the sign. The pattern for the number 13 stored as an `int` would be

```
00000000000000000000000000001101
```


`short` -32,768 to 32,767 (16 bits)

`int` -2,147,483,648 to 2,147,483,647 (32 bits)

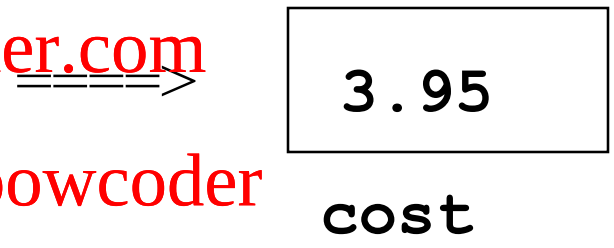
`long` -9,223,372,036,854,775,808 to
9,223,372,036,854,775,807 (64 bits)

Type double

The type **double** is used to store a real number , i.e. a number with a decimal point. It is stored in memory in a different format from an **int**.

You can do arithmetic on a **double**.


```
double cost = 3.95;
```




```
cost * 0.10
cost / 2
cost + cost * 0.1
```

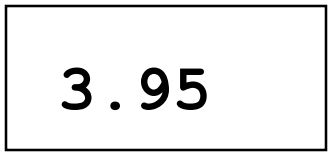
Type double

The type **double** is used to store a real number , i.e. a number with a decimal point. It is stored in memory a different format from an **int**.

You can do arithmetic on a **double** .


```
double cost = 3.95;
```

=====>




```
cost * 0.10          → 0.395
cost / 2             → 1.975
cost + cost * 0.1    → 4.345
```

Type char

A **char** is a character, i.e. a bit pattern you can produce by pressing a key (or a combination of keys) on a keyboard.

Examples :

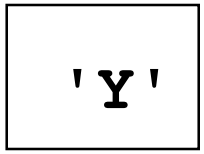
note the single quotes


```
'a'   'A'   '3'   '?'   '!'
```



```
char response = 'Y';
```

=====>



response

Note : you cannot add '3' and '2' and get '5'

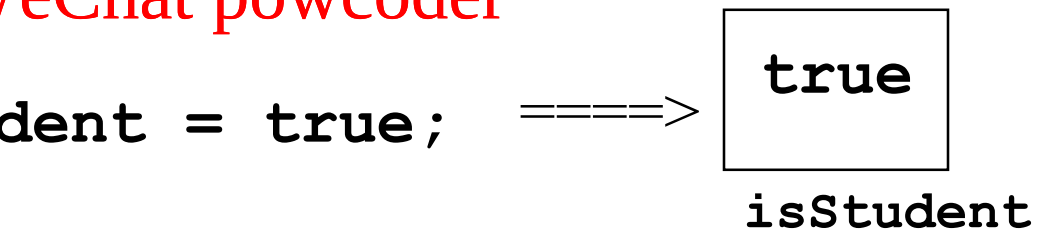
(ie. *characters are not the same as integers*)

Type `boolean`

A `boolean` variable can have a value of only `true` or `false`. This could be stored as one bit (but actually isn't).

For example, a `Person` object may or may not represent a student. We could have an attribute of `Person` called `isStudent` which would be either `true` or `false` for a particular `Person` instance.


```
boolean isStudent = true;
```



Type `String`

note the *double quotes*

`String` is a "special" *object* type.

A `String` is a collection of `chars` (e.g. `"Sally"`).

```
String name = "Sally";
```


A `String` value is always written in *double quotes*.
You can have an empty `String`, shown as `""`.

Examples of *behaviour* of a `String` object include:

- **give you its value in upper/lower case**
- **tell you how long it is (how many characters)**
- **give you the character at a specified position**

String concatenation

The plus operator (+) has different meanings depending on the *type* of its operands.

- If both operands are numbers, it represents addition.
- If both operands are strings, the strings are concatenated (joined together) into a single string.
- If one operand is a string and the other is not then the other operand is converted to a string and then the strings are concatenated.


```
return "Student is: " + name;
```

Type compatibility during assignments

In Java, every variable has a type. Java will not allow a value of the wrong type to be put into a variable.

If we write, for example,

```
int aNumber = "3";
```


BlueJ will give the error message

incompatible types -found java.lang.String but expected int

```
String aName = true;
```

produces the error message

incompatible types - found boolean but expected java.lang.String.

Displaying information

Output on the screen is produced by invoking (calling) the `println` method of the `System.out` object that is built into the Java language. For example:

```
System.out.println("# Ticket");
```


This statement displays the string between the quotes and then moves the cursor to the next line.


```
System.out.println("# " + price + " cents.");
```


The '+' is used to construct a single string, which is then displayed.

Another option is the `print` statement which is identical to `println` but it does not move the cursor to the next line.

Example : Displaying information

Note how the strings are concatenated!

```
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;
}
```