7SENG003W Advanced Software Design

Classes and objects in Java and UML

This week's aims

- Understand definitions of object and class
- Introduce to UML, a visual OO language used for designing OO applications
- How to represent object and class
  - In Java
  - In UML
- Types and classes in Java
- How to relate:
  - Objects
  - Classes

Simple Application – Inventory Control

- Requirements:
  - Keeps track of parts in a manufacturing assembly and how they are used in constructing assemblies. We'll use the following terminology
    - **Parts** : individual, physical objects
    - **Assemblies** : complex structures made up out of parts, and possibly structured into subassemblies
  - The program could
    - Maintain catalogue information: what kinds of parts are there?
    - Maintain inventory information: how many are in stock?
    - Record structure of manufactured assemblies
    - Provide operations on assemblies
      - Cost - based on cost of component parts
      - Print "parts explosion"

–We'll use this application for example code etc.

The Object Model

- Underlying all OO methodologies is a shared understanding of objects, classes etc…. this is encapsulated in the **OBJECT MODEL**
- Object-Oriented design methods and object-oriented programs are based on the same assumptions about the structure of software
  1. The basic architecture of a system is assumed to be a set of classes - STATIC
  2. At run-time, instances of these classes interact to perform various tasks – i.e., a running program is a network of objects - DYNAMIC
- The Object Model provides the foundation for OO design and programming
  - It defines what is a class, object etc.
  - How are objects connected?
  - How are classes related?

What is an Object?

- First, we need to be clear what we mean by "object" – although there is not one clear definition
- One possible definition - an object is a "tangible entity that exhibits some well-defined behaviour" (Booch, 94)
  - This can be applied to any kind of object (real-world etc.)
  - It can also apply to abstract entities, such as numbers
- A software object models some part of reality within a software application in a way that makes it a distinguishable entity with well-defined behaviour
  - What do we mean by "well-defined behaviour"?
  - An object is defined by its behaviour – it is what it does…
- A software object generally accepted to have three characteristics
  - State – the data held by an object and its value(s) at any point in time
  - Behaviour – what the object can do
  - Identity – how to uniquely refer to an object from among all the other objects that exist

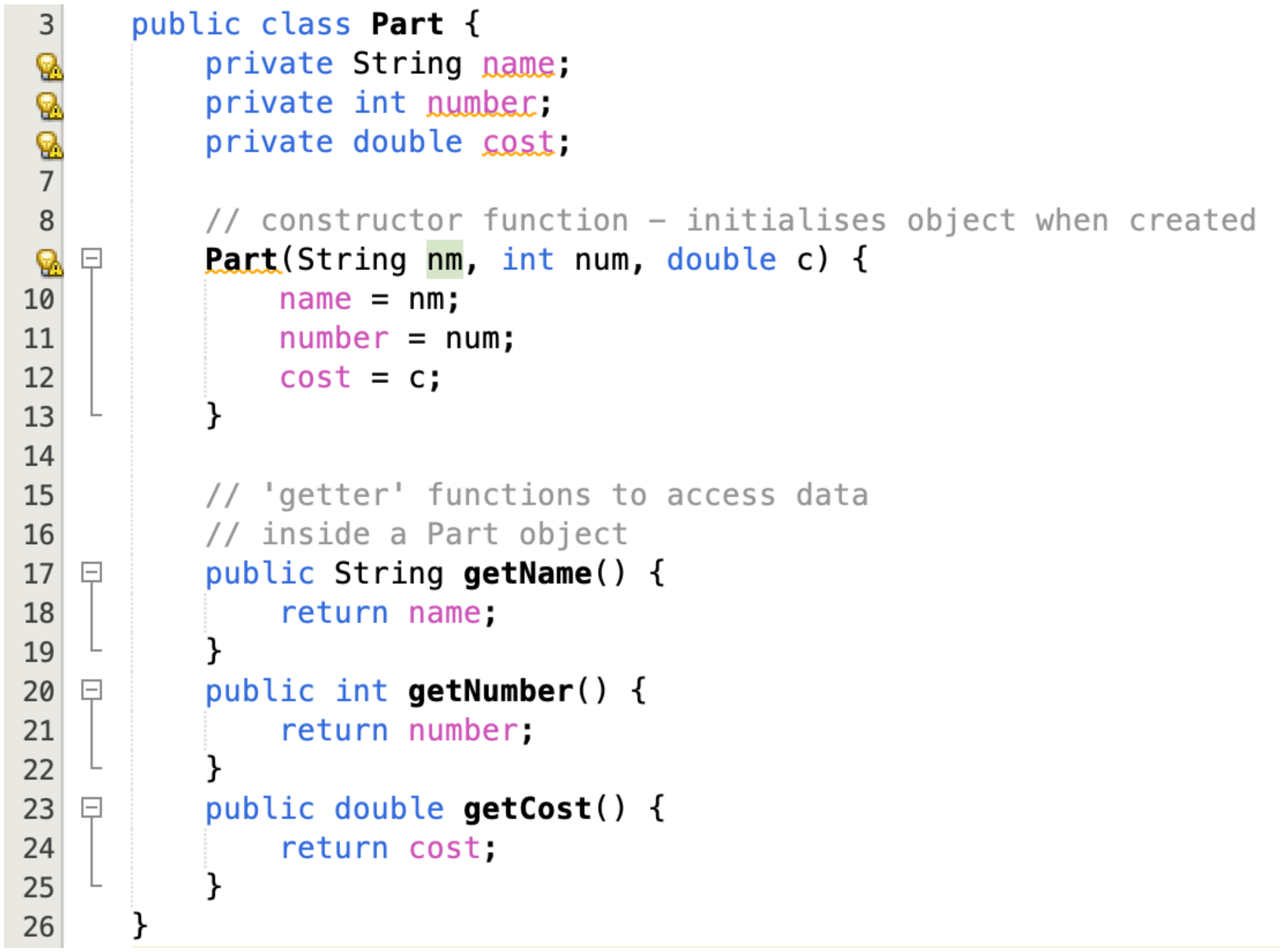
What is a Class

- A class is really three different concepts – or rather, there are three different perspectives on the meaning of a class
- A class can be viewed as:
  - A **template** for creating objects, by specifying its attributes and its operations. An object created from a class will have all the attributes and operations laid out in the class declaration
  - A **type**, i.e, an indication of which contexts a value can be used in – in the same way that `int` or `float` are types
  - A definition of a **set** of objects that exhibit common structure and behaviour – any object that is a member of a class has the same structure and behaves in the same way as other objects in that class

– Note these are not necessarily synonymous – the class of an object defines a type of the object, but a type of an object is not necessarily its class

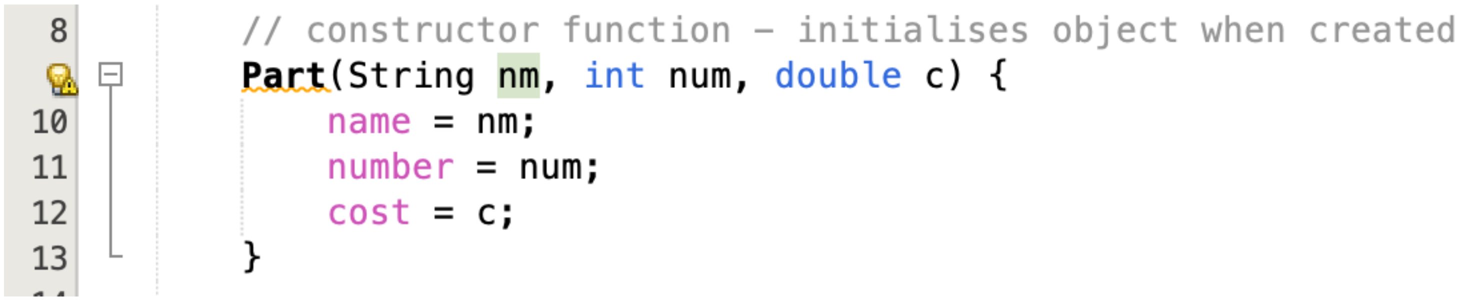
Classes in C#

- Classes in Java are defined in much the same way as C# and C++ - with minor differences
- Note access specifiers: ‘private’, ‘public’ which determine who can ‘see’ or use the things in the class



```
public class Part {
    private String name;
    private int number;
    private double cost;

    // constructor function - initialises object when created
    Part(String nm, int num, double c) {
        name = nm;
        number = num;
        cost = c;
    }

    // 'getter' functions to access data
    // inside a Part object
    public String getName() {
        return name;
    }
    public int getNumber() {
        return number;
    }
    public double getCost() {
        return cost;
    }
}
```

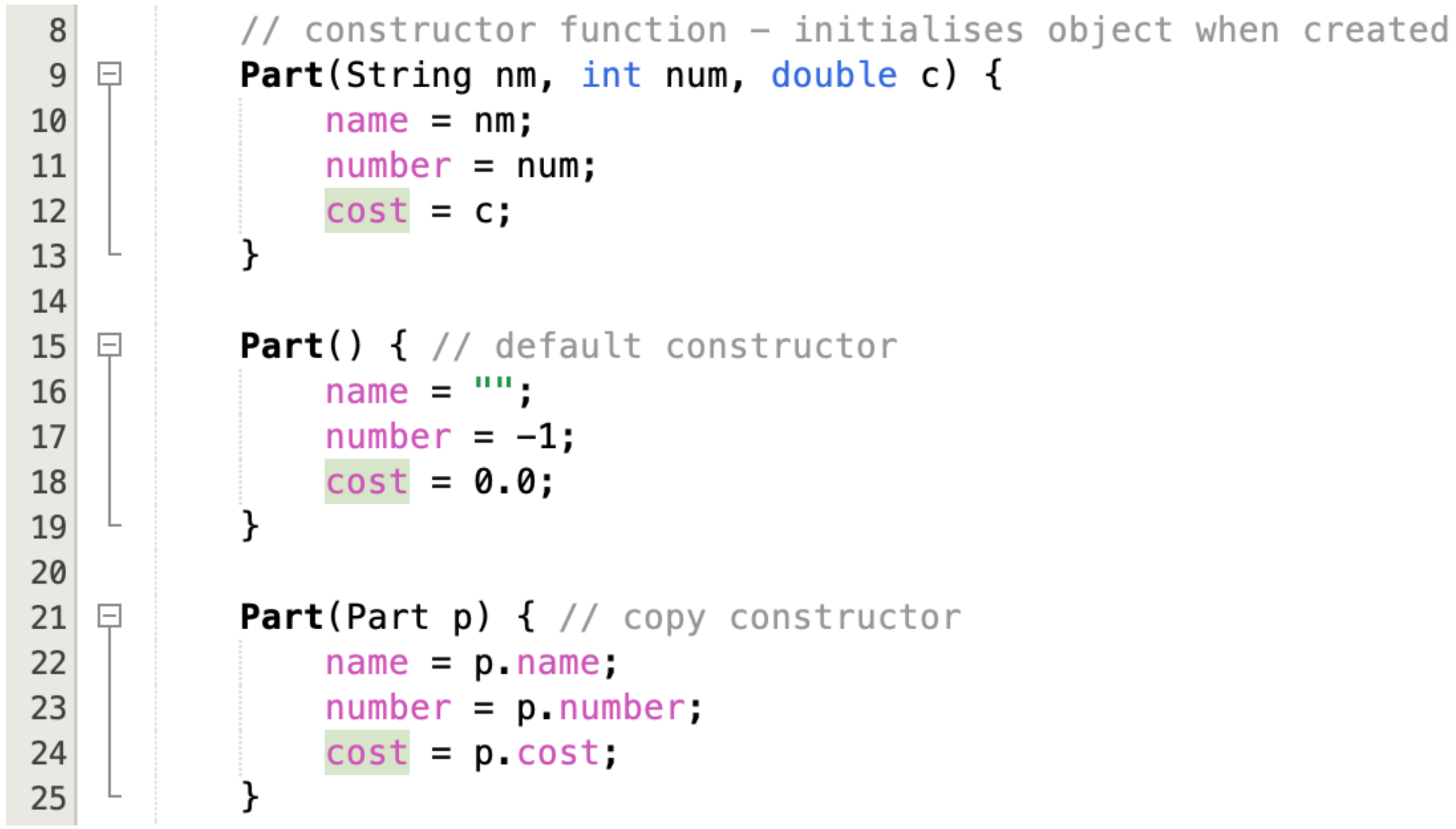
Constructors



```
    // constructor function - initialises object when created
    Part(String nm, int num, double c) {
        name = nm;
        number = num;
        cost = c;
    }
```

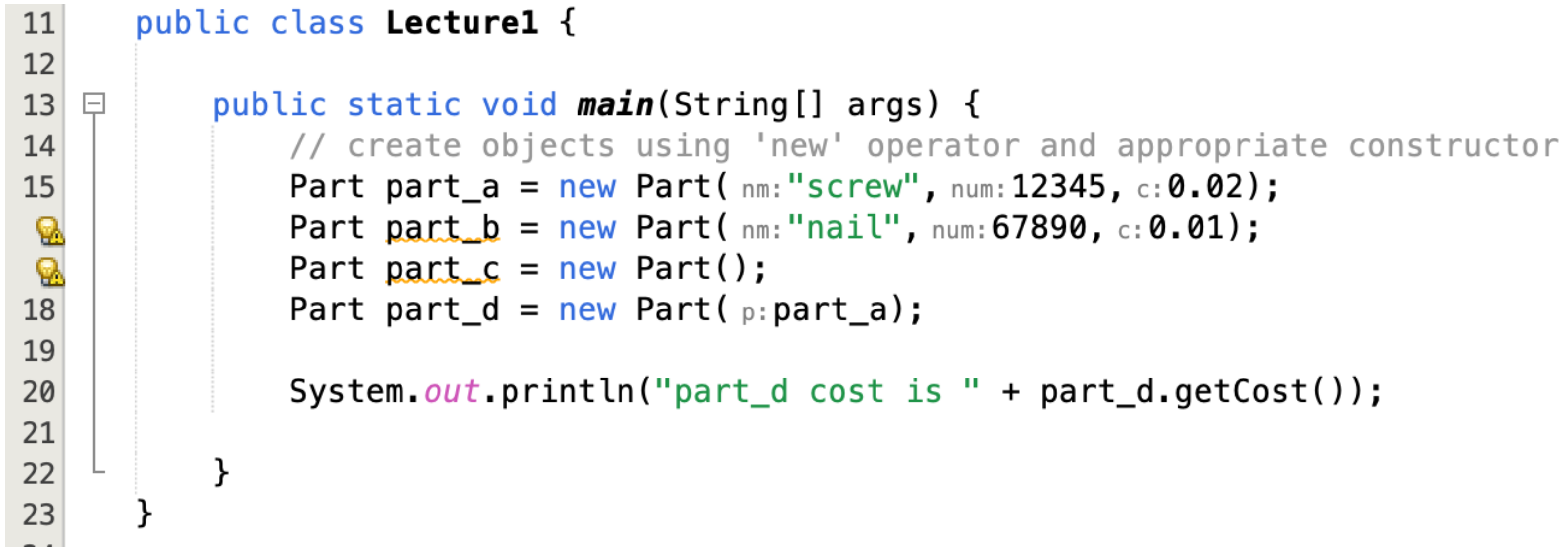
- Used in object creation – when new object requested, constructor is called with appropriate arguments to initialize new object
- A class can have many constructors to create an object
- Constructor with no arguments is called the **default** constructor

Constructors for the Part class



```
    // constructor function - initialises object when created
    Part(String nm, int num, double c) {
        name = nm;
        number = num;
        cost = c;
    }

    Part() { // default constructor
        name = "";
        number = -1;
        cost = 0.0;
    }

    Part(Part p) { // copy constructor
        name = p.name;
        number = p.number;
        cost = p.cost;
    }
```

Creating objects



```
public class Lecture1 {

    public static void main(String[] args) {
        // create objects using 'new' operator and appropriate constructor
        Part part_a = new Part( nm:"screw", num:12345, c:0.02);
        Part part_b = new Part( nm:"nail", num:67890, c:0.01);
        Part part_c = new Part();
        Part part_d = new Part( p:part_a);

        System.out.println("part_d cost is " + part_d.getCost());

    }
}
```

- To create an object from a class, we need to call the appropriate constructor with the right arguments
- First part object created using 3-argument constructor
- Second part object created from first using copy constructor
- **new** operator – allocates memory space dynamically to create object and calls constructor to set up object in that space

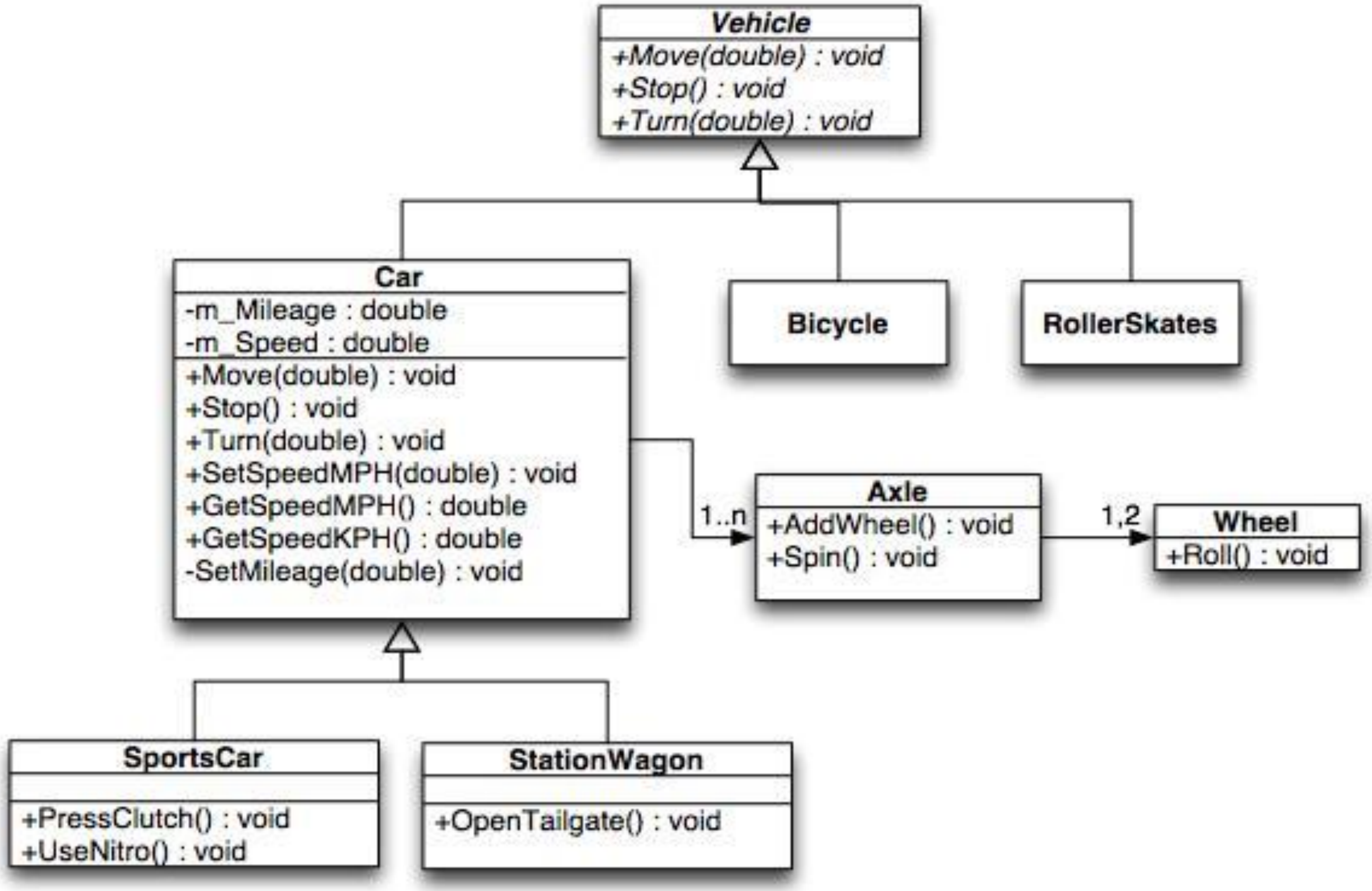
Objects and classes in UML

- When we design OO applications, we want to use a programming-language independent notation for the design
- UML (Unified Modelling Language) is perhaps the most popular and significant OO design notation currently in use. It is a unification of three earlier methods
  - OMT (James Rumbaugh)
  - The Booch Method (Grady Booch)
  - OOSE (Ivar Jacobson)
- UML was initially developed at the Rational Corporation, but has been submitted to the OMG (Object Management Group) for approval as a standard. UML 2.1.2 was released in Nov 2007 (see Object Management Group website: www.omg.com)
- Unlike its predecessors, UML is **not a methodology by itself**
  - UML is a (visual) **language**
  - It does **not define** a process for development
  - The “Rational Unified Process” is a complementary development process, from Rational, but which does not form part of UML

The Nature of UML

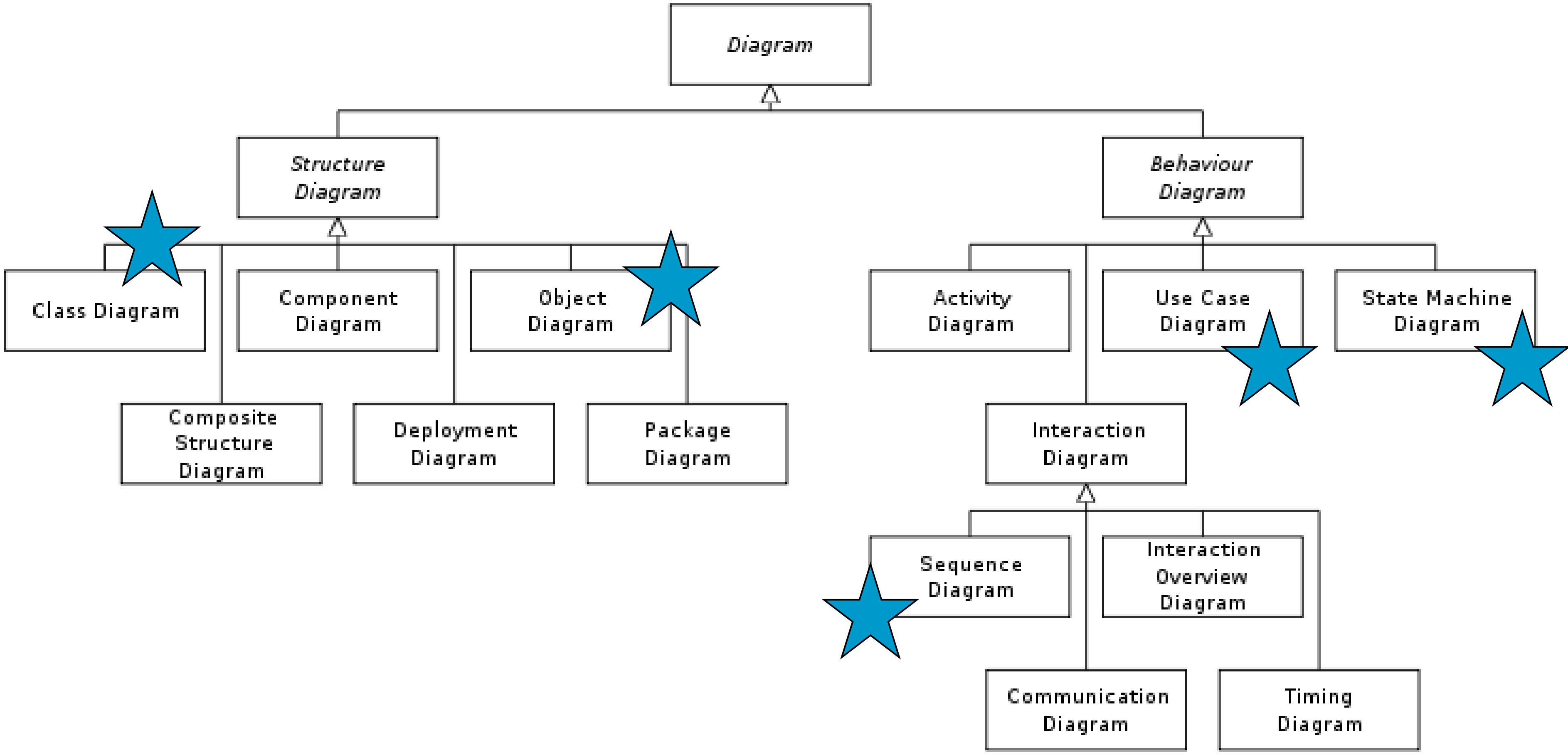
- UML is a graphical language. This means that the information being recorded is presented using diagrams rather than in textual form. Many people, and many engineering disiplines, find graphical presentations of structural information easier to handle than textual descriptions
- This does not mean that using UML is simply a question of drawing pictures. Typically, many different diagrams will be used to document the design of a system, and they will all have to be consistent with each other
- It is helpful to imagine that there is a single database of information describing the system being designed, and that the various diagrams are simply providing convenient views of some data in this database
  - CASE tools are often used to help create UML diagrams – offer automatic model verification (Rational Rose)

Example class diagram



(From http://spacecraft.sourceforge.net/doc/api/ProgrammingRefresher.html)

Different Types of UML Diagram

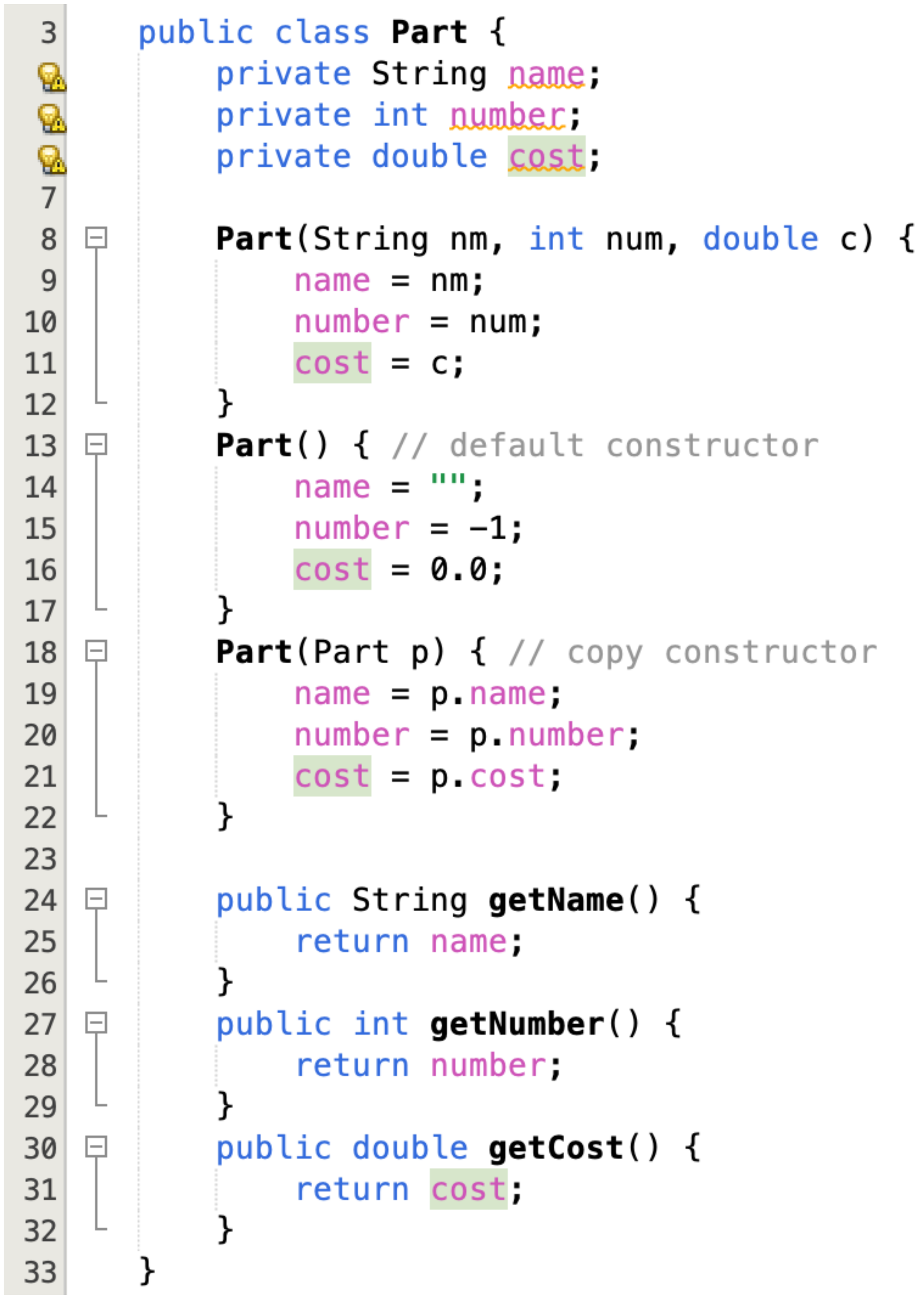


★ = we'll cover this diagram on this module

Models and diagrams

- UML makes a distinction between models and diagrams
- Models are what you create as an abstraction of your system
- Diagrams are convenient representations of model information
  - But they are NOT the model itself
  - Just “windows” into the model
  - Each diagram shows part of the model
  - Different diagram types represent different aspects or perspectives of the model (e.g, runtime behaviour / static system structure etc.)

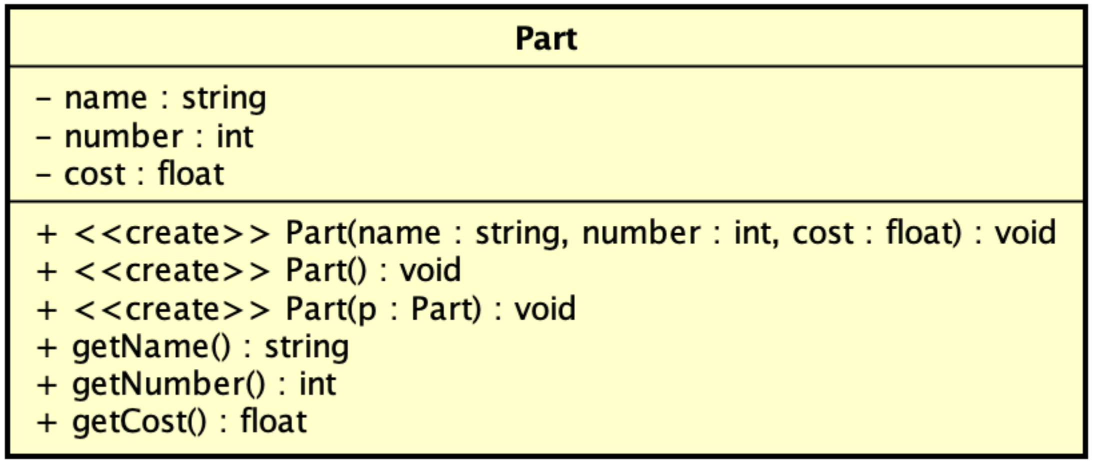
The Part class (reminder)



```java
public class Part {
    private String name;
    private int number;
    private double cost;

    Part(String nm, int num, double c) {
        name = nm;
        number = num;
        cost = c;
    }
    Part() { // default constructor
        name = "";
        number = -1;
        cost = 0.0;
    }
    Part(Part p) { // copy constructor
        name = p.name;
        number = p.number;
        cost = p.cost;
    }

    public String getName() {
        return name;
    }
    public int getNumber() {
        return number;
    }
    public double getCost() {
        return cost;
    }
}
```

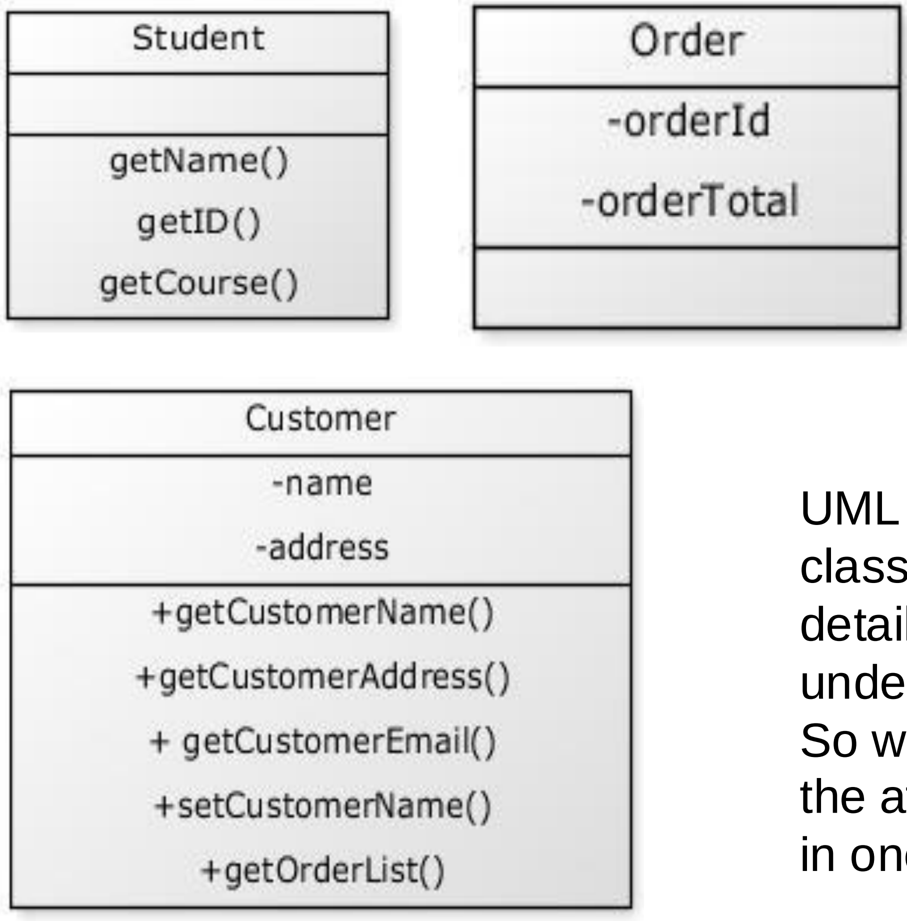
Representing the Part class in UML

- Here is the part class represented in UML:


Part
– name : string
– number : int
– cost : float
+ <<create>> Part(name : string, number : int, cost : float) : void
+ <<create>> Part() : void
+ <<create>> Part(p : Part) : void
+ getName() : string
+ getNumber() : int
+ getCost() : float

- Classes are represented in UML as rectangular icons, divided into three sections: name, attributes, methods
- Note the use of the create stereotype (<<create>>) to distinguish constructors from ordinary methods
- We also show accessibility
  – - (private), + (public), # (protected)

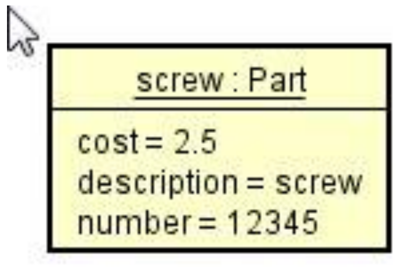
More example classes



UML allows us to build up classes incrementally, adding detail as we refine our understanding of the model. So we don't have to add all the attributes and methods in one go.

UML: Object Notation

- UML provides the following notation to show individual objects:



  - The object's name and the name of its class name are shown underlined in the top compartment
  - Attribute names and values in the lower compartment
  - Operations are not shown
  - The object name is used to distinguish different instances of the same class (often the name of the variable storing the object…)
  - Note: can't show how the object was created (dynamically or not…)
- Conventions
  - Class names start with upper-class letters
  - Object names start with lower-class letters

Digression No. 2: classes vs types

- What's the difference between a class and a type? Is there one?
- We can write `int i = 5;`
- … and we can write `Part p = new Part("bolt",98765,0.03);`
- One uses a (built-in) type, the other uses a class – but they look the same
- So are classes and types the same thing? Answer: yes and no
- A type describes an abstraction and the services that the abstraction offers
  - A type is therefore a kind of *contract* – it specifies what services a value offers
  - E.g., `int` says you can add, multiply, integer divide, take remainder etc. from such values
- A class defines how a particular (user-defined) type or abstraction is implemented
  - We don't need to do this for built-in types because – their implementation is built-in (already done for us)
- A value can have just one class but *many* types (polymorphism) – as we will see

Types

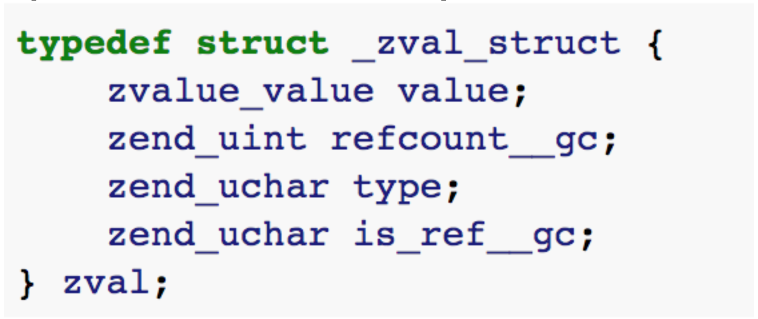
- Some programming languages are known as “strongly-typed” (or that they have “static typing”) – e.g, C/C++, Java, C#, Haskell
- Generally, this means the programmer has to add type annotations to the code, e.g.,

```
int x;
string s;
```

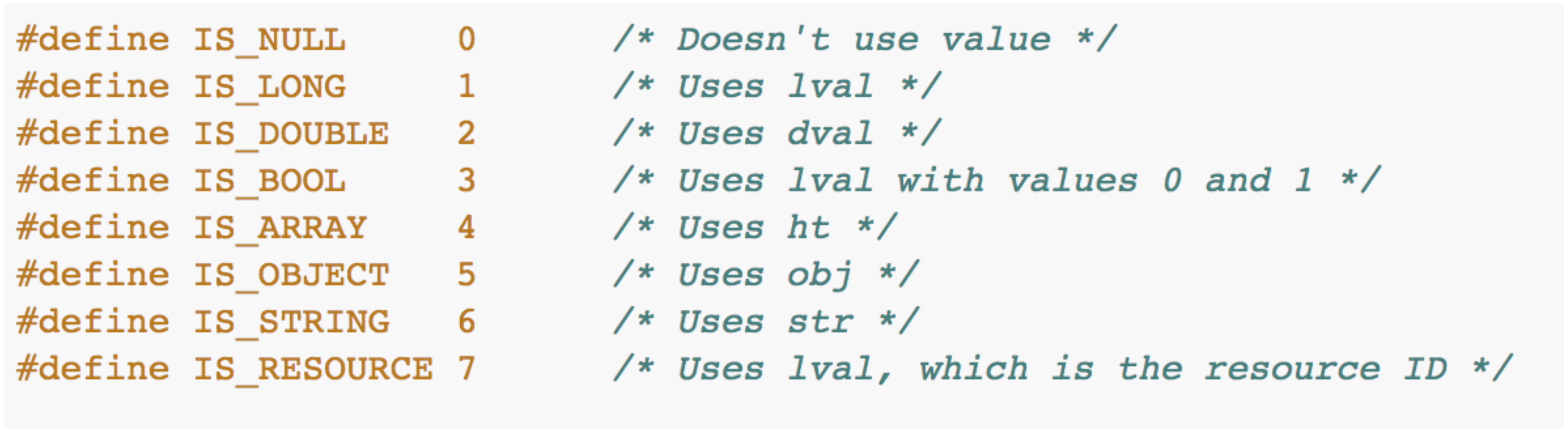
- A type defines a kind of contract – what services you can expect of a value of that type
- Types are conventionally used to ensure program correctness. Checking values are used according to their type helps spot (some) errors, for example.
- Typed programming languages normally involve a compilation phase before program execution (either to machine code or bytecode) – among other things, this allows type checking to happen before program executes

“Untyped” languages

- (Almost) all programming languages use types – even the “untyped” ones (e.g, PHP, Python) – which are not really untyped…
- The programmer doesn’t add type annotations, but types are still there beneath the surface – because types are necessary to know how data is laid out in a variable
- E.g., PHP – uses ”type tags” in zvals to know how to interpret bit pattern in memory allocated to a variable



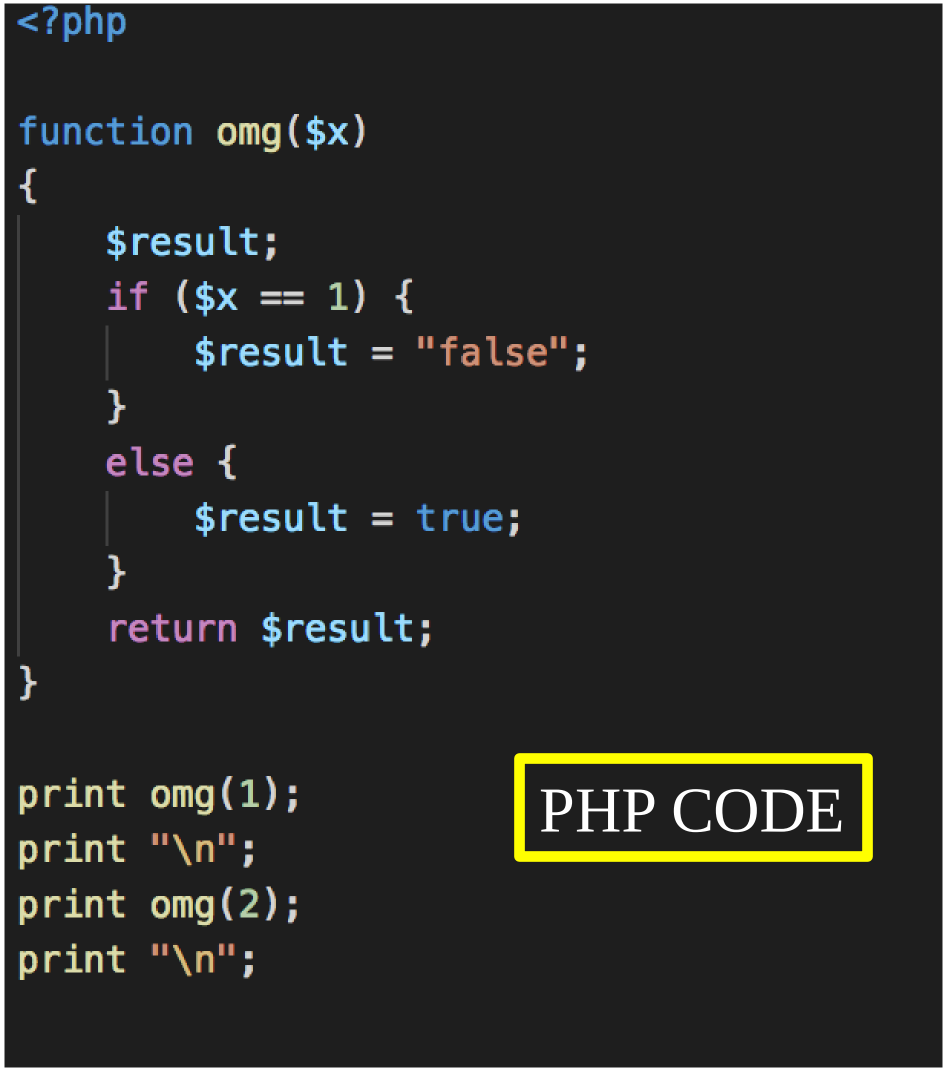
```c
typedef struct _zval_struct {
    zvalue_value value;
    zend_uint refcount__gc;
    zend_uchar type;
    zend_uchar is_ref__gc;
} zval;
```



```c
#define IS_NULL     0       /* Doesn't use value */
#define IS_LONG     1       /* Uses lval */
#define IS_DOUBLE   2       /* Uses dval */
#define IS_BOOL     3       /* Uses lval with values 0 and 1 */
#define IS_ARRAY    4       /* Uses ht */
#define IS_OBJECT   5       /* Uses obj */
#define IS_STRING   6       /* Uses str */
#define IS_RESOURCE 7       /* Uses lval, which is the resource ID */
```

Types and program structure

- Types also help in providing or enforcing structure. For example, in **PHP**, a programming language with no explicit types, you can write things like this:

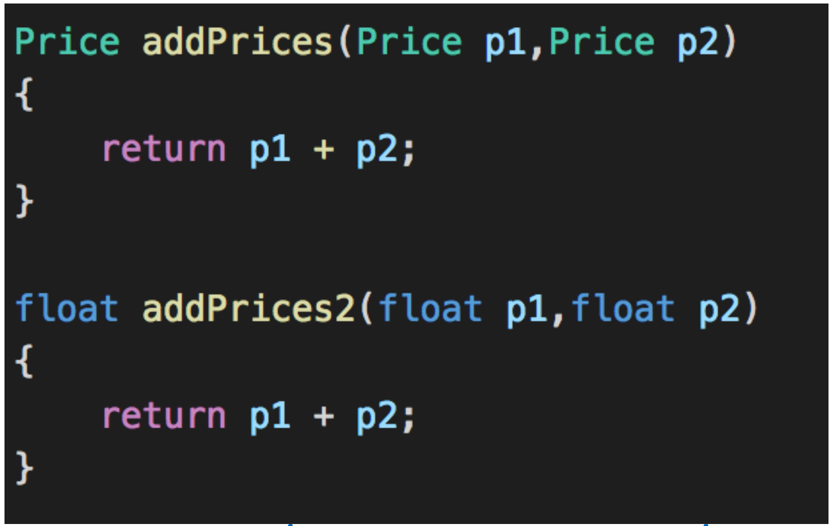


```php
<?php

function omg($x)
{
    $result;
    if ($x == 1) {
        $result = "false";
    }
    else {
        $result = true;
    }
    return $result;
}

print omg(1);
print "\n";
print omg(2);
print "\n";
```

- Languages without types require much more discipline to structure well
- Without types, you are never quite sure what kind of value (string? boolean?) a variable might hold

Good type design helps spot errors

- Pierce, *Types and Programming Languages*, 2nd Ed. MIT Press, 2002
- Good type design can help with error checking
- E.g., which helps detect errors better?



```
Price addPrices(Price p1,Price p2)
{
    return p1 + p2;
}

float addPrices2(float p1,float p2)
{
    return p1 + p2;
}
```

- https://gist.github.com/simoncourtenage/0eb97c4c7735a23d4c927e9a9db4c957
- https://en.wikipedia.org/wiki/Mars_Climate_Orbiter (space probe crash caused by mistaking imperial measurements for metric)

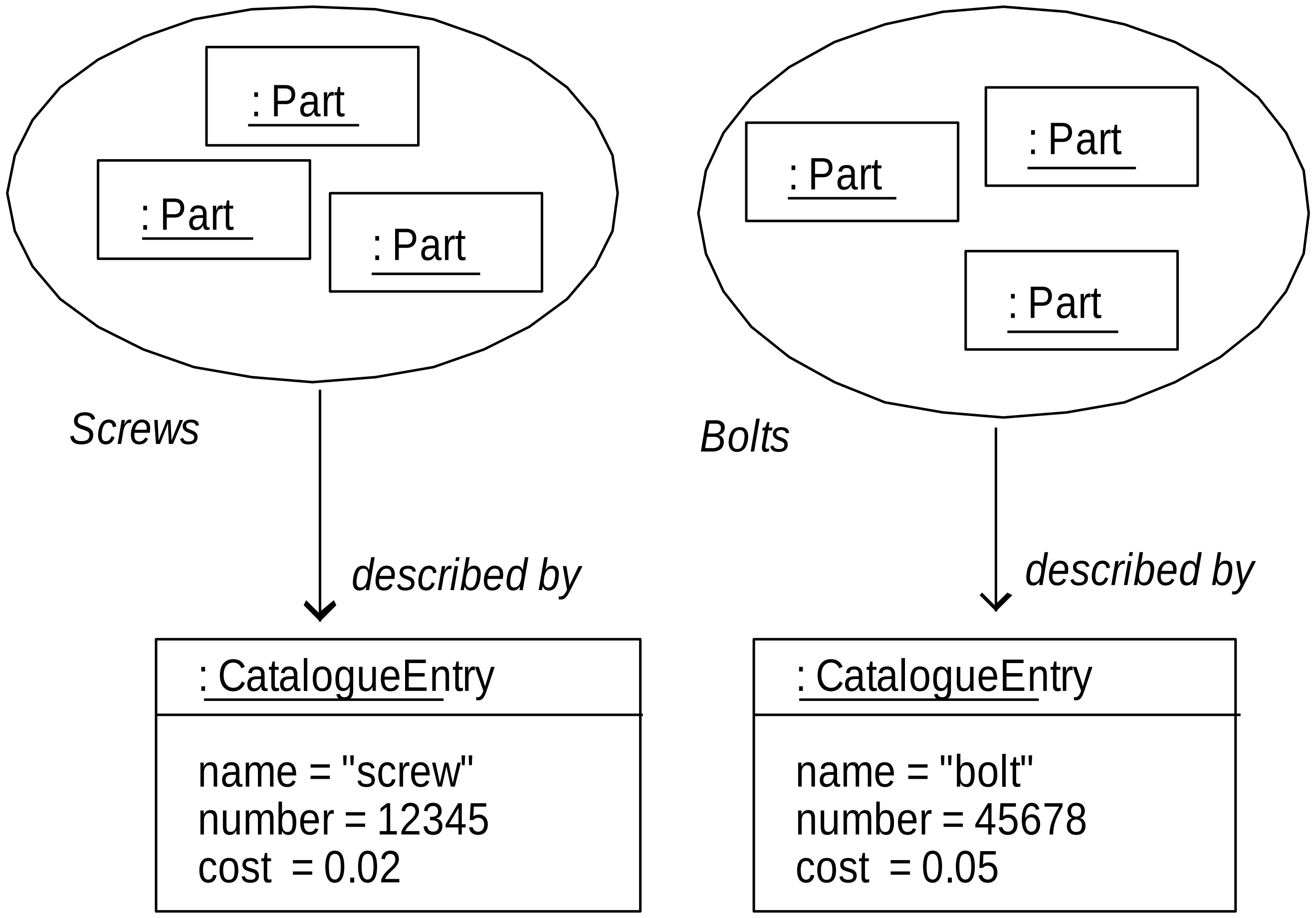
Connecting objects: Problems with the Part class

- If we have 100,000 screws, the design would represent them by 100,000 screw objects, each storing the ID number of the part, its description and its price

- Two major problems with this major
  - Redundancy
    - Storing the same data 100,000 times will waste significant amounts of storage
  - Maintainability
    - If for example the price of a screw changed, there would be 100,000 updates to perform in order to update the system. AS well as being time-consuming, this will make it very hard to ensure that the system's data is accurate and consistent.

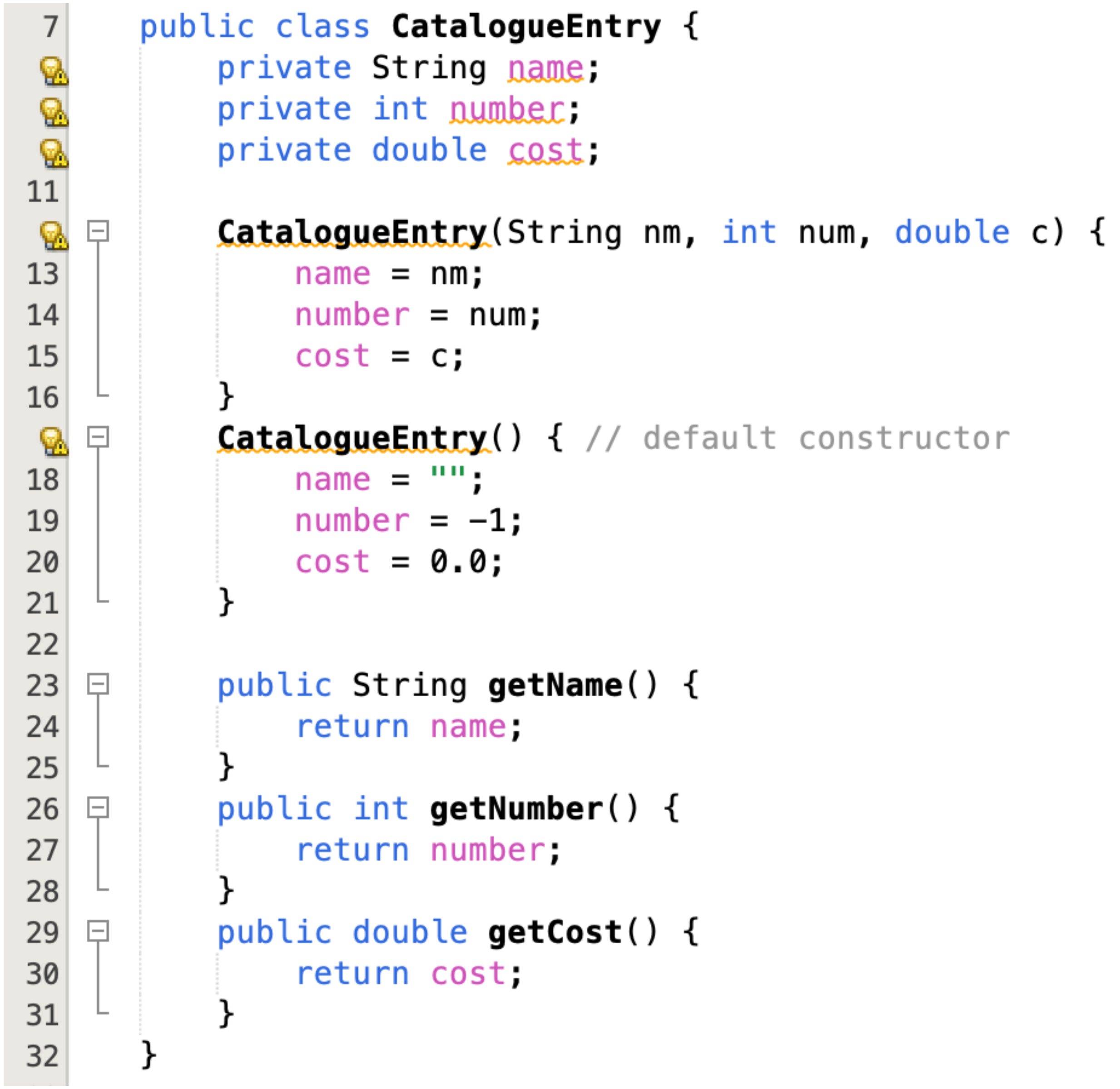
Solution

- Move the shared information out of the part objects into a new object that describes a particular kind of object – I.e, a kind of catalogue entry – and connect a part to a catalogue entry object that describes it



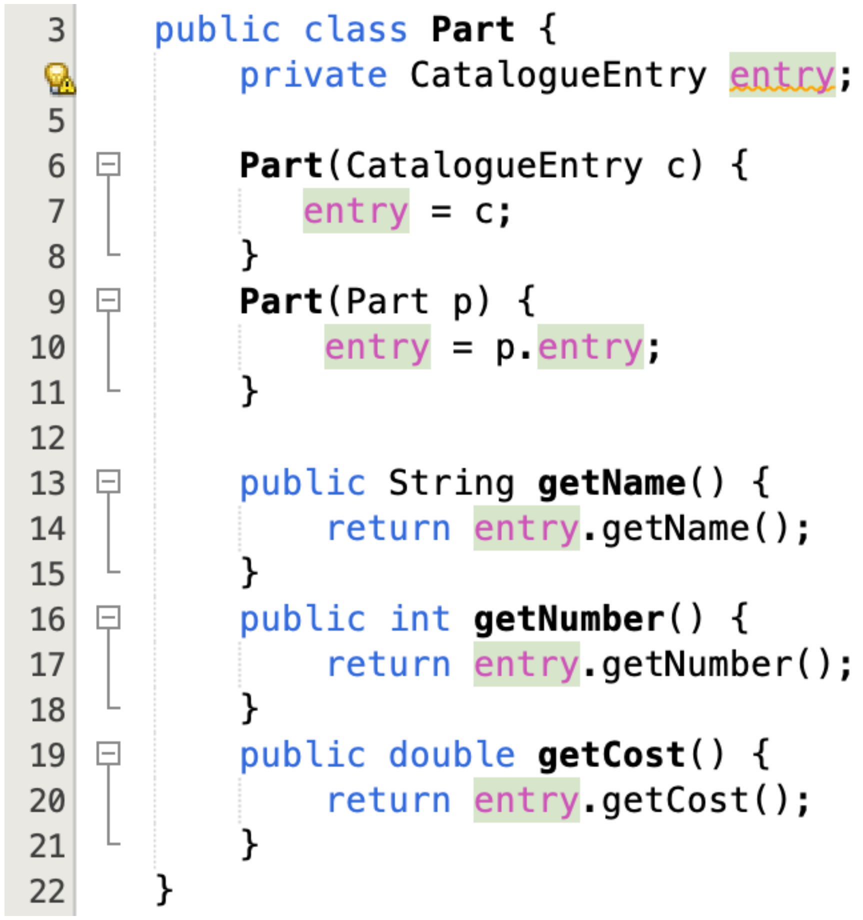
Connecting objects

- To implement this new design, the attributes in the part class would be removed into the catalogue entry class, and each part object would contain a reference to the relevant catalogue entry object



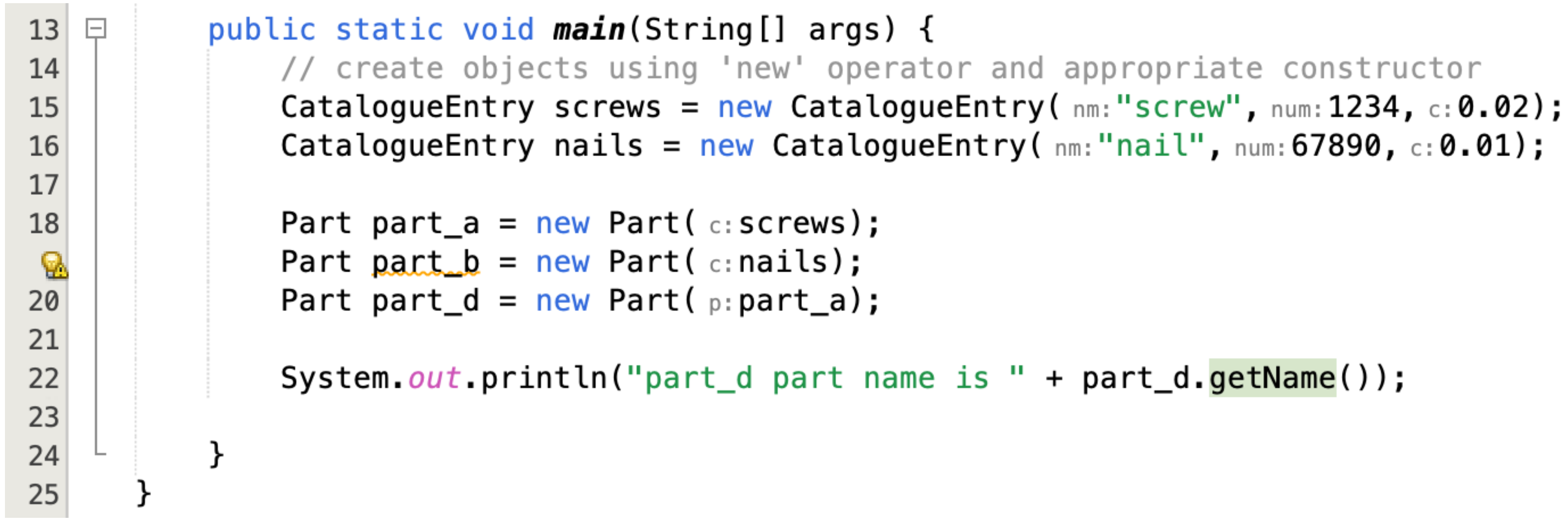
```java
public class CatalogueEntry {
    private String name;
    private int number;
    private double cost;

    CatalogueEntry(String nm, int num, double c) {
        name = nm;
        number = num;
        cost = c;
    }
    CatalogueEntry() { // default constructor
        name = "";
        number = -1;
        cost = 0.0;
    }

    public String getName() {
        return name;
    }
    public int getNumber() {
        return number;
    }
    public double getCost() {
        return cost;
    }
}
```

The updated Part class and how to use it



```java
public class Part {
    private CatalogueEntry entry;

    Part(CatalogueEntry c) {
      entry = c;
    }
    Part(Part p) {
        entry = p.entry;
    }

    public String getName() {
        return entry.getName();
    }
    public int getNumber() {
        return entry.getNumber();
    }
    public double getCost() {
        return entry.getCost();
    }
}
```

Main function – connecting Parts and CatalogueEntry objects

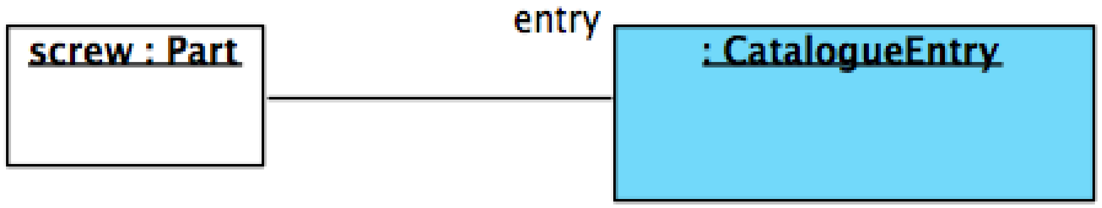


```java
    public static void main(String[] args) {
        // create objects using 'new' operator and appropriate constructor
        CatalogueEntry screws = new CatalogueEntry( nm:"screw", num:1234, c:0.02);
        CatalogueEntry nails = new CatalogueEntry( nm:"nail", num:67890, c:0.01);

        Part part_a = new Part( c:screws);
        Part part_b = new Part( c:nails);
        Part part_d = new Part( p:part_a);

        System.out.println("part_d part name is " + part_d.getName());

    }
}
```

Redundancy is now removed because all Part objects that are screws get their information from a single CatalogueEntry object

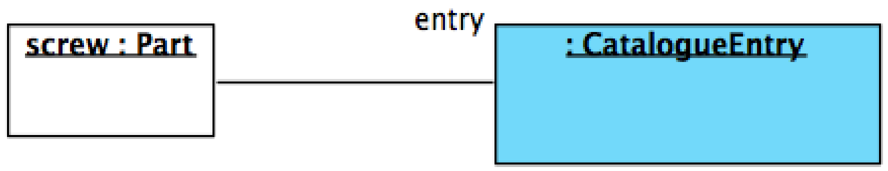
Links

- At run-time, every Part object will contain a reference to a catalogue entry object. This enables it to call the operations of the catalogue entry object, for example to find out its cost.
- In UML, a run-time connection between objects is known as a ***link***, and is represented by a line joining the two objects
- A link between two objects represents a kind of communications channel, over which methods calls can be made – the channel is based on object identity



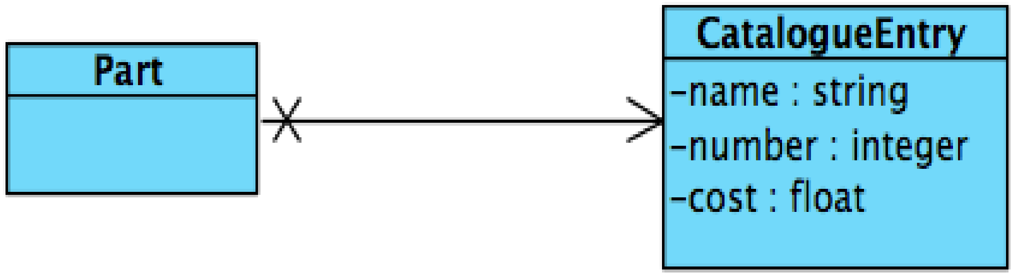
Representing objects – communication diagrams

- The diagram on the previous slide is a simple *communication diagram*. An object diagram shows some objects and links between them at a given moment in time.
- An object diagram is a kind of pictorial memory map of a running program, but rather than showing actual memory addresses, objects and pointers are shown graphically
- Links store data
  - Links store data structurally. The fact that a part is of a particular kind is represented by the link between the two objects, not a data item in the part object
  - Although entry is a data member of the Part class, it is not shown as an attribute on the object diagram. The data stored in that attribute is shown instead as a link. The data member name can be used as a label on the link, as shown.



Associations

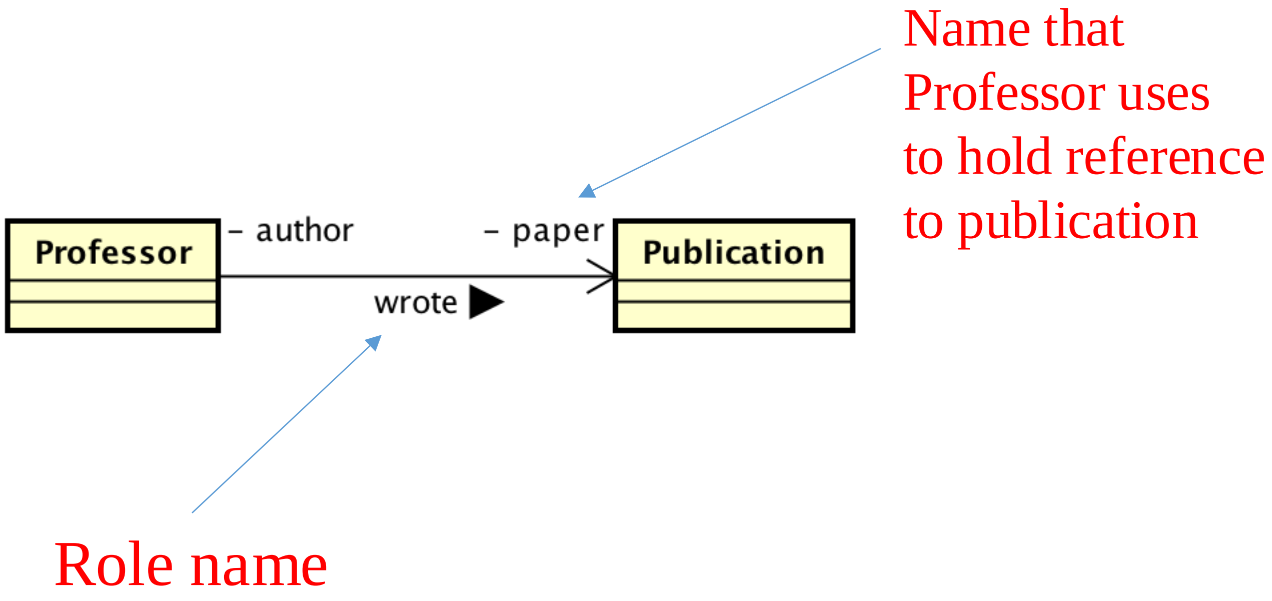
- In the same way that objects are instances of classes, links are thought of as being instances of things known as *associations*. An association is basically a relationship between classes: it asserts that instances of the classes can be linked at run-time
- An association is shown in UML as a line joining two classes. If the links can only be navigated in one direction, this can be shown as a property of the association



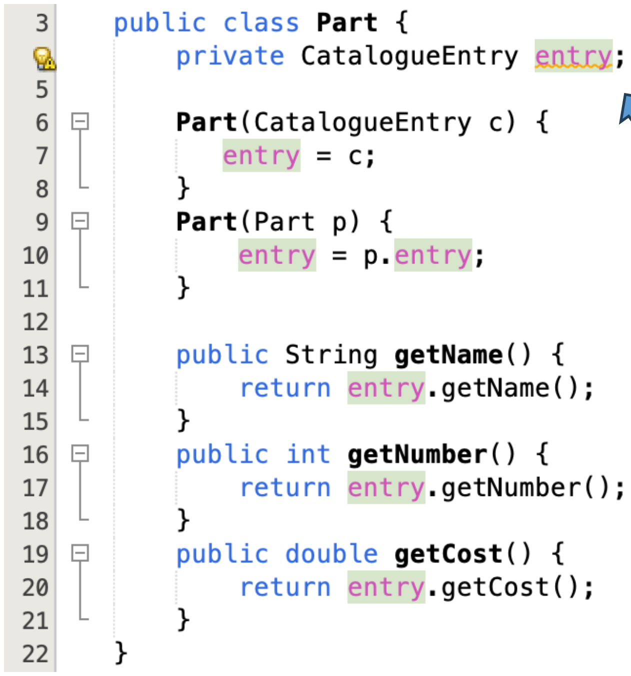
- Associations are not explicitly supported by C++/C#/Java. They are implemented by defining suitable data members in the relevant classes

Association role names

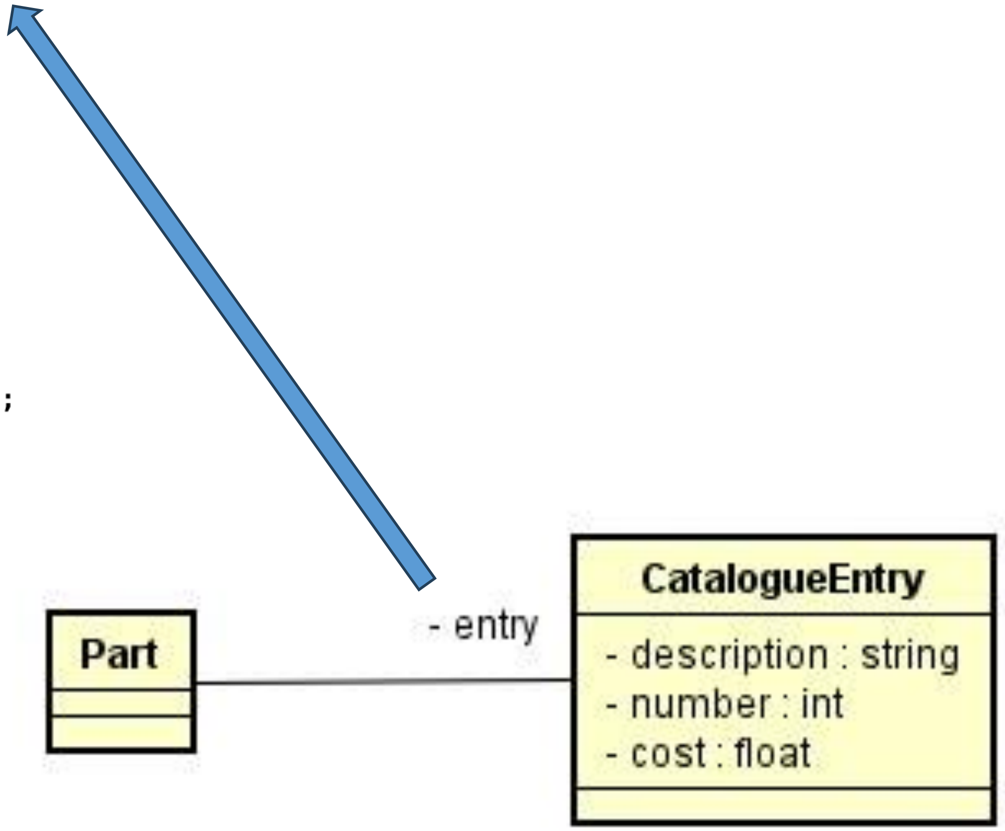
- Role names:
  - “A role name explains how an object participates in the relationship. Each object needs to hold a reference to the associated object or objects. The reference is held in an attribute value within the object. When there is only one association then there is only one attribute holding a reference.” (from UML Bible)



Associations are structural relationships



```
public class Part {
    private CatalogueEntry entry;

    Part(CatalogueEntry c) {
        entry = c;
    }
    Part(Part p) {
        entry = p.entry;
    }

    public String getName() {
        return entry.getName();
    }
    public int getNumber() {
        return entry.getNumber();
    }
    public double getCost() {
        return entry.getCost();
    }
}
```



What kind of role name could be used?