Object Oriented Software Design

Constructors, Implementation hiding, Inheritance

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

October 28, 2010

Outline

Outline

Constructor

- Every class has a *constructor*
  - A special function that is called when the object is created with **new**
  - The function must have the same name as the class, and can have any number of parameters
  - You can also specify more than one constructor, with different parameter lists
  - The constructor without parameters is called *default constructor*
- By default:
  - If you do not specify any constructor, the compiler will automatically generates a default constructor
  - If you do specify a constructor (with any number of parameters), the default constructor is not generated

Constructor example

SimpleConstructor.java

```
class Rock {
  Rock() { // This is the constructor
    System.out.println("Creating Rock");
  }
}

public class SimpleConstructor {
  static Test monitor = new Test();
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new Rock();
    monitor.expect(new String[] {
      "Creating Rock",
      "Creating Rock",
      "Creating Rock",
      "Creating Rock",
      "Creating Rock",
    });
  }
}
```

Counting the Rocks

Exercise

Add another constructor that takes an integer and a string to name the Rock.

Exercise

By using a static member, count the number of rocks that have been created till now by either of the two constructors

Initialization

- data members inside a class can be initialized:
  - on-line, when declaring the data member
  - inside the constructor
  - inside *initialization clauses*
- the second approach gives much more flexibility, as the initialization can depend on the arguments
- a *initialization clause* is a block of code that you can insert anywhere inside the class, and that contains code for initializing the data members

Initialization

- You can initialize a data member with a constant, or with the value of a function:
  - The function can take as arguments initialized variables

Correct:

```
class CInit {
  int i = f();
  int j = g(i);
  //...
}
```

Wrong:

```
class CInit {
  int j = g(i);
  int i = f();
  //...
}
```

- It is also important to understand the initialization order: `./examples/05.java-examples/OrderOfInitialization.j`
  - Data members are initialized in the same order they are defined;
  - After that, the constructor is called

Static initialization

- When static data members are initialized?
  - Only if necessary, just before a static method is called, or the first object is created
  - `./examples/05.java-examples/StaticInitialization.jav`

Initialization clauses

- You use also *initialization clauses*

```
class Cups {
  static Cup c1;
  static Cup c2;
  static {
    c1 = new Cup(1);
    c2 = new Cup(2);
  }
  ...
```

```
public class Mugs {
  Mug c1;
  Mug c2;
  {
    c1 = new Mug(1);
    c2 = new Mug(2);
    System.out.println("c1 & c2 initialized");
  }
  Mugs() {
    System.out.println("Mugs()");
  }
  ...
```

- These clauses are executed in the same order they are written in the file
  - This syntax is necessary for *anonymous inner classes* (see later)

Outline

Overloading

- You can define many member functions with the same name
  - For example, you can have many constructors
  - The same rule applies to *normal* member functions
- However, functions with the same name must have different argument list
  - The list may differ in the number of arguments and/or in their types, and/or in the order
- The function name, along with the argument list, is called *method signature*
- You can define as many methods as you want in a class, but they must have all different signatures
- `./examples/05.java-examples/Overloading.java`, `./examples/05.java-examples/OverloadingOrder.java`

Promotion and overloading

- Pay careful attention to overloading with primitive types
  - Promotion must be taken into account
- The rule is: the compiler will try to use the method whose parameters are “closer” to the ones you are giving
  - First, it looks for an exact match of types
  - Then it tries to promote until it finds a match
  - If there is no match, the compiler issues an error
- You can force this behavior by appropriate casting
- `./examples/05.java-examples/PrimitiveOverloading.ja`

Overloading and return type

- You cannot overload on return types
  - The return type is not part of the signature
- This is because the *correct* return value cannot be automatically inferred by the compiler

```
void fun();
int fun();
...

f();
```

Which version the compiler should call?

this

- the **this** keyword is used to refer to the object reference from within the class

```
class Apricot {
  int data;
  void pick() { data = 25; }
  void pit() { pick(); }
}
```

≡

```
class Apricot {
  int data;
  void pick() { this.data = 25; }
  void pit() { this.pick(); }
}
```

- As you can see, in most of the cases you do not need **this**, because it is implicit
- it is used mainly:
  - when you want to return a reference to the current object (`./examples/05.java-examples/Leaf.java`)
  - when you want to call another constructor (`./examples/05.java-examples/Flower.java`)

Multiple constructors

- When you have multiple constructors, it is likely that they will share some code
  - Some of the data members wll need to be initialized anyway,
  - in some cases, some of these initialization do not depend on the specific parameters
- When the same code is repeated several times, there is always a danger of introducing some mistake
  - Consider for example the case in which we have to change some code in the constructor
  - if this is code is duplicated in all constructors, we **must** remember to correct all constructors, otherwise we introduce a subtle bug!
  - **Hint:** try to avoid duplication of code
- **Remember:** the constructor call must be the first thing you do, or you'll get a compiler error message.

Outline

Cleaning up

- In C++, it is possible to write a *destructor* function for every class, that is called when the object memory is destroyed
- In Java, **there is no destructor** function
  - the reason is that the main need for destruction is to reclaim memory (as we will see in the C++ language)
  - Java provides its own automatic way of reclaiming memory through the garbage collector
- However, it is sometimes necessary to perform some additional task when an object is not used anymore, before it is completely deleted by the garbage collector
  - Suppose that you are using your own way to allocate memory, not on the heap but on a special storage
  - Since the GC knows nothing about it, it is not going to free that special memory
- To solve this problem, you can write a **finalize()** method

finalize

- The **finalize()** is called by the GC before deleting the object
  - At the first pass, if the GC finds an object that should be deleted, it first calls its **finalize()** method
  - At the second pass, the GC actually deletes the object
- **Warning:** it is possible that the GC is never executed
  - for example when your program uses little memory, there is no need to call the GC
- If the GC is never executed, the **finalize()** is never called!
- Thus, there is a **huge** difference between the **finalize()** method in Java and the destructor in C++

More on GC and finalize

- You can suggest the execution of the GC by calling **System.gc()** at some point
- You can suggest the finalization of all pending objects by calling **System.runFinalization()**
- Nevertheless, **finalize()** is not the right place where to put generic clean-up code
  - You should do the clean-up by yourself (e.g. close open files), with appropriate methods
  - GC and **finalize()** are only for releasing memory!
- **finalize()** can also be used for debugging (for example to check if proper clean-up has been done)
- `./examples/05.java-examples/TerminationCondition.ja`

Array initialization

- It is possible to initialize array of objects using the curly braces syntax:

```
public class ArrayInit {
  public static void main(String[] args) {
    Integer[] a = {
      new Integer(1),
      new Integer(2),
      new Integer(3),
    };
    Integer[] b = new Integer[] {
      new Integer(1),
      new Integer(2),
      new Integer(3),
    };
  }
}
```

- The second for is useful for defining functions with variable-length arguments
  - for example, the **Test.expect()** function
  - Also `./examples/05.java-examples/VarArgs.java`

Outline

Package

- A package is what becomes available when you use the import keyword to bring in an entire library:
- If you want to use a given class in a package, you can import the package, import the single class **or** use the full path to the class:

```
// import the whole package
import java.util.*;
...
// import the single class
import java.util.ArrayList;
...
// use the full path
java.util.ArrayList al = new java.util.ArrayList();
```

Compilation Unit

- A source file (ending with .java) is a compilation unit
  - Inside it, there can be at most on class declared **public**, that must have the same name as the file
  - There can be other non-public classes
    - these are not available from outside
  - After compilation, you get a .class file
  - All class files can be *packaged* in a .jar file (java archive)
- A library is a group of class files, or a jar file
  - To name the library/package, you can state the following instruction at the beginning of a .java file

```
package mypackage;
public class MyClass {
  // . . .
```

Your first custom tool library

- You can build you own custom tool library
  - Consider the following class, used to reduce the amount of printing:

```
public class P {
  public static void rint(String s) {
    System.out.print(s);
  }
  public static void rintln(String s) {
    System.out.println(s);
  }
}
```

- You can put it into a package **tools** that is located somewhere in your hard disk
  - I suggest you use a specific directory structure: for example `oosd/tools`

```
package oosd.tools;
```

- Now, this example uses the package:
  `./examples/05.java-examples/UseToolsExample.java`

Package problems

- From “Thinking in Java”

  It’s worth remembering that anytime you create a package, you implicitly specify a directory structure when you give the package a name. The package must live in the directory indicated by its name, which must be a directory that is searchable starting from the CLASSPATH. Experimenting with the package keyword can be a bit frustrating at first, because unless you adhere to the package-name to directory-path rule, you’ll get a lot of mysterious run-time messages about not being able to find a particular class, even if that class is sitting there in the same directory.

- It’s so much true!

Public, private, protected

- Let's recall the access specifiers
  - **public** means that the member can be accessed by every other class. The member is *part of the* **interface**
    - `./examples/05.java-examples/dessert/Cookie.java`, `./examples/05.java-examples/Dinner.java`
  - **private**: the member is only accessible by members of the same class. The member is *part of the* **implementation**
    - It is possible to *hide* the constructor in this way: `./examples/05.java-examples/IceCream.java`
    - Can you explain why this may be useful?
  - **protected**: the member is accessible from the class members, and from all the members of derived classes. Again, the member is part of the **implementation**

Package access

- If you do not specify any access specifier (**public private**, **protected**), then the default is **package public**
  - it means that the member (or class) can be accessed by all classes of the same package
  - but not from outside the package, for them the member is **private**
- A class can be **public**, or **package public**

Outline

Reusing classes

- The old way of reusing code was to copy and paste from one file to another

Reusing classes

- The old way of reusing code was to copy and paste from one file to another
  - Can you tell me why this is bad?

Reusing classes

- The old way of reusing code was to copy and paste from one file to another
  - Can you tell me why this is bad?
- To be able to efficiently re-use code, we should never change existing code

Reusing classes

- The old way of reusing code was to copy and paste from one file to another
  - Can you tell me why this is bad?
- To be able to efficiently re-use code, we should never change existing code
  - if the code is tested and works well

Reusing classes

- The old way of reusing code was to copy and paste from one file to another
  - Can you tell me why this is bad?
- To be able to efficiently re-use code, we should never change existing code
  - if the code is tested and works well
  - if it has a clear interface

Reusing classes

- The old way of reusing code was to copy and paste from one file to another
  - Can you tell me why this is bad?
- To be able to efficiently re-use code, we should never change existing code
  - if the code is tested and works well
  - if it has a clear interface
  - and if we use it according to the interface

Reusing classes

- The old way of reusing code was to copy and paste from one file to another
  - Can you tell me why this is bad?
- To be able to efficiently re-use code, we should never change existing code
  - if the code is tested and works well
  - if it has a clear interface
  - and if we use it according to the interface
  - everything should work fine

Reusing classes

- The old way of reusing code was to copy and paste from one file to another
  - Can you tell me why this is bad?
- To be able to efficiently re-use code, we should never change existing code
  - if the code is tested and works well
  - if it has a clear interface
  - and if we use it according to the interface
  - everything should work fine
- Reusing code is difficult!

Reusing classes

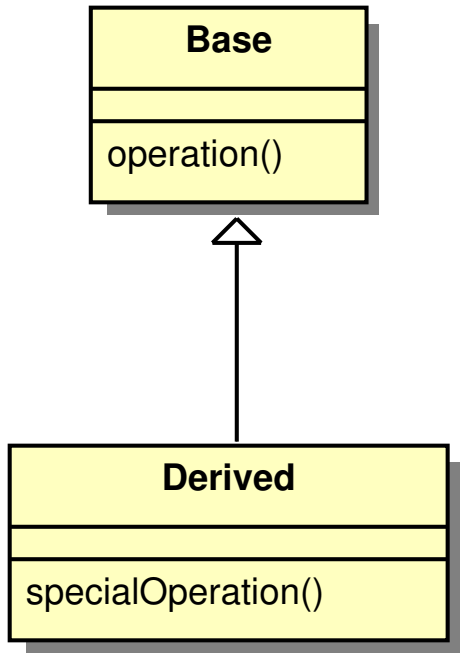
- The old way of reusing code was to copy and paste from one file to another
  - Can you tell me why this is bad?
- To be able to efficiently re-use code, we should never change existing code
  - if the code is tested and works well
  - if it has a clear interface
  - and if we use it according to the interface
  - everything should work fine
- Reusing code is difficult!
- Even more difficult is to write *reusable code*

Composition and inheritance

- There are two ways of reusing code:
  - **Composition**: you define an object of the class that you want to reuse in your code, and call its interface
  - **Inheritance**: you extend existing classes, modifying their behavior, or adding additional behaviors
- The first one we have already seen. We will analyze it better as we encounter it during the rest of this lecture
- Inheritance is a new thing, let's start understanding how to do that

Inheritance

- With inheritance, it is possible to specialize a class
  - Inheritance models the *is-a* relationship
    - the base class is the general
    - the derived class is the specialized
  - We say that: **Derived** *is-a* **Base**
  - however, **Base** is not a **Derived**



Inheritance Syntax

- To implement that in Java:

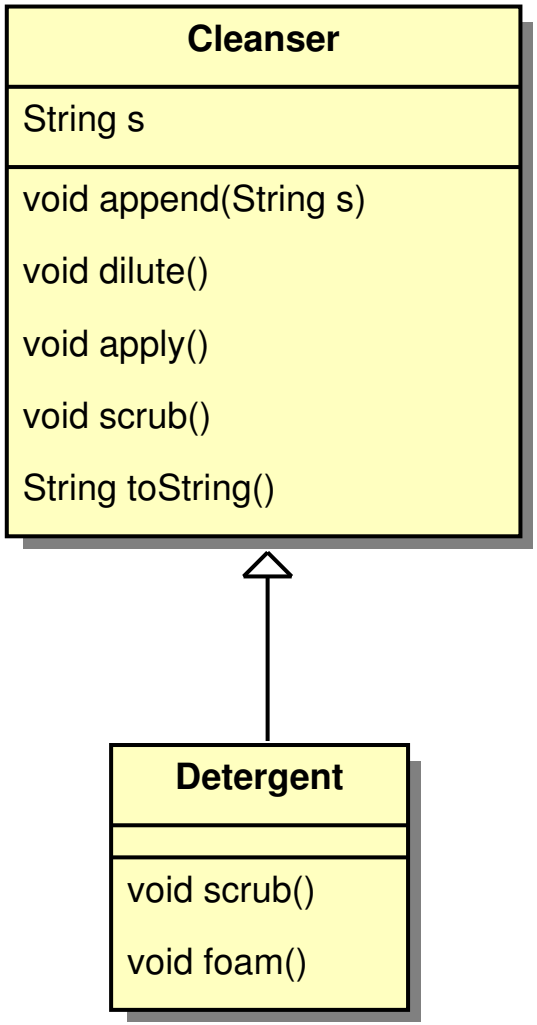
```
class Base { ... }

class Derived extends Base { ... }
```

- An example:
  `./examples/05.java-examples/Detergent.java`

Example

- There are many things to notice:
  - **Cleanser** is the base class, **Detergent** is the derived class
  - In the derived class, we can use **super** to refer to the base class members
  - In `Derived.scrub()`, we call `super.scrub()` to execute the function `Cleanser.scrub()`



main

- As you can see in the previous example, the `main` function is defined for both classes
  - This is perfectly legal in Java, and it is used to test classes in isolation
  - When you execute the JVM, you must specify a class name, and that class must contain the `main` function
- Therefore, the previous example can be executes as
  - `java Cleanser` (and the `Cleanser.main()` is executed)
  - `java Detergent` (and the `Detergent.main()` is executed)

toString

- consider the following code:

```
MyClass s = new MyClass();
...
String s = "this is " + s;
```

- Java tries to convert s into a string, so that it can be appended to `s`
  - to do this, Java looks for the `toString()` method in `MyClass`
  - if there is not such a method, the standard method of class `Object` is used that returns a string containing the class name and the address in memory of the object
  - otherwise the method is *overloaded* by `MyClass` (as with `scrub()`).

Initialization

- How does the object of a derived class get constructed?
  - First, the constructor of the base class is called: then constructor of the derived class is executed
  - From a layout point of view, you can imagine the derived object as containing a special **subobject** of the base class that needs to be initialize

Initialization

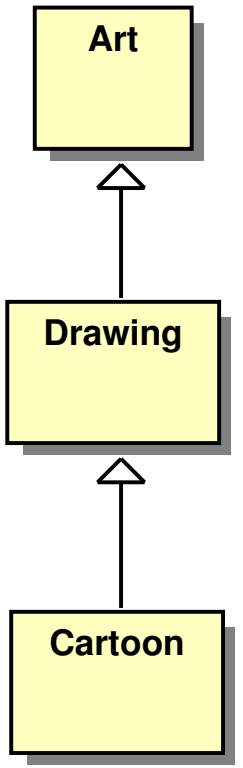
- How does the object of a derived class get constructed?
  - First, the constructor of the base class is called: then constructor of the derived class is executed
  - From a layout point of view, you can imagine the derived object as containing a special **subobject** of the base class that needs to be initialize
- If the base class has a default constructor, the Java compiler automatically inserts a call to it (**super()**) in the derived class constructor

Initialization

- How does the object of a derived class get constructed?
  - First, the constructor of the base class is called: then constructor of the derived class is executed
  - From a layout point of view, you can imagine the derived object as containing a special **subobject** of the base class that needs to be initialize
- If the base class has a default constructor, the Java compiler automatically inserts a call to it (**super()**) in the derived class constructor
- However, it is often necessary to directly call the constructor of the base class
  - This can be done by calling the base class constructor through the `super` keyword

Example of initialization

- Here is an example:
  `./examples/05.java-examples/Cartoon.java`
  - The UML class diagram
    - Notice that we can hide members, we are not interested to them right now
  - In this case, the constructor is implicitly called by Java
  - Exercise: modify the code to put a non-default constructor with arguments in **Art**
  - Warning: the call to the base-class constructor **must be the first thing** you do in the derived-class constructor.

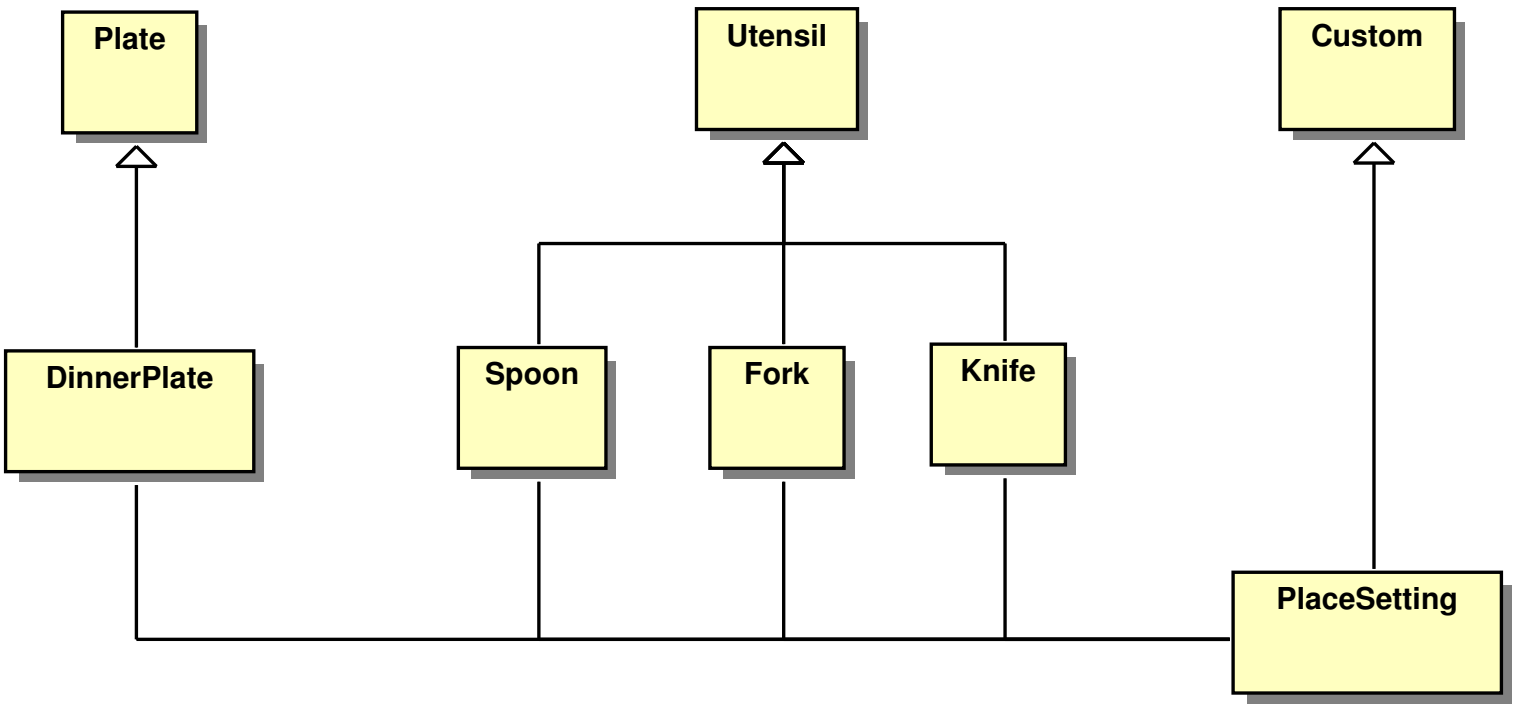


More on initialization

- It is important to point that classes are loaded in memory at the point of their first use
  - Every class is in a separate .class file
  - Java loads them only at the point where they are actually used
  - Also initialization takes place at the point of the first use (this is especially valid for static members
- An example:

Combining Inheritance and Composition

- A more complex example:
  `./examples/05.java-examples/PlaceSetting.java`



Protected

- The keyword **protected** allows a member to be accessed by subclasses
  - *protected* also implies *package public*: a protected member can be accessed by other classes in the same package
    - Java is very confusing about this package feature!
- An example: `./examples/05.java-examples/Orc.java`

Upcasting

- Upcasting means that you can use an object of the derived class as it were an object of the base class, and everything works fine
- Example: `./examples/05.java-examples/Wind.java`
  - In the example, function `static void tune(Instrument i)` takes a reference to an `Instrument`
  - However, it is called later as `Instrument.tune(flute)`, i.e. passing a reference to a `Wind` object
- **Hint:** when should you use inheritance?
  - ask yourself: do I need to upcast (i.e. treat the derived class as the base class)?

Outline

The final keyword

- Java has one keyword **final** with many slightly different meanings
- In general, you can read **final** as “it cannot be changed”
  - However, the exact meaning depends on its usage

Final data members

- When applied to primitive data members, **final** means *constant*
  - a data member that is static and final has only one single storage that cannot be changed
  - the compiler is allowed to optimize calculations with static final objects that are initialized at compile time
- When applied to references, **final** means that the reference is constant (it always refers to the same object), but the object is allowed to change its internal values
  - Java has no ways of making an object to be constant (unlike C++)
- Example:
  `./examples/05.java-examples/FinalData.java`
- Another example: blank finals
  `./examples/05.java-examples/BlankFinal.java`
- Remember: You are forced to perform assignments to finals either with an expression at the point of definition of the field or in every constructor.

Final arguments

- When an argument is final, it means that you cannot change it inside the function
  - Again, for references it means that the reference is final, but you can change the values inside the object
- This feature is completely unuseful
- Remember: this is different from **const** in C++!
  - In C++ you cannot modify a const object

Final methods

- A final method cannot be overridden in derived classes
  - All private methods in a class are implicitly final
  - A final method can be optimized as in-line code

Final methods

- A final method cannot be overridden in derived classes
  - All private methods in a class are implicitly final
  - A final method can be optimized as in-line code
- Unfortunately, derived class can define methods with the same signature as private methods in the base class
  - this is legal, because the private method is not part of the interface, therefore **it is not seen** outside
  - and hence, another method with the same signature can be defined in the derived classes, without causing any conflict
  - This is badly confusing! (but this is Java)
  - See `./examples/05.java-examples/Beetle.java`

Final methods

- A final method cannot be overridden in derived classes
  - All private methods in a class are implicitly final
  - A final method can be optimized as in-line code
- Unfortunately, derived class can define methods with the same signature as private methods in the base class
  - this is legal, because the private method is not part of the interface, therefore **it is not seen** outside
  - and hence, another method with the same signature can be defined in the derived classes, without causing any conflict
  - This is badly confusing! (but this is Java)
  - See `./examples/05.java-examples/Beetle.java`
- Notice that this is not *overriding*, cause overriding only occurs between methods that are part of the interface

Final classes

- A final class cannot be extended
  - No derived classes for it
- All methods in a final class are implicitly final
- It seems sensible to make classes final for whatever reason
- However, pay attention to this: sometimes the performance you gain with final are completely negligible with respect to the usefulness of extending an existing class

Outline

Exercises

Exercise

Take one of the examples with base-derived class, and declare the constructor of the base class private. Then create an object of the derived class. Can you explain what will happen? Check your hypothesis by compiling and running the example.

Exercise

Take one of the examples with base-derived class, and declare a function protected. Then create a function with the same signature in the derived class, but declare it public. What happens? Check your hypothesis by compiling and running the example.

Exercises

Exercise

- Write a program with one base class **A** and two derived classes **B** and **C**. Put a object counter in **A** that counts how many objects of any of the three classes have been created, and a method to get this number.
- Then, do the same in every derived class. In **B**, put a counter to count how many object of **B** have been created, and in **C** a counter of objects of **C**. Write appropriate **get** members.
- Also, write a method **dispose()** in the three classes, which must be called by the user before destroying the objects, that decreases the appropriate counters.
- Finally, write a test for this program, and check out its correctness.