

Algorithmics 3

Final words on Java

December 6th, 2006

Object-Oriented Programming

You have seen

- ▶ procedural programming
- ▶ object-oriented programming.

Object-Oriented Programming

You have seen

- ▶ procedural programming
- ▶ object-oriented programming.

Q What's the difference ?

Object-Oriented Programming

You have seen

- ▶ procedural programming
- ▶ object-oriented programming.

Q What's the difference ?

Q What is Object-Oriented Programming all about ?

Philosophy

Object-Oriented Programming is all about

- ▶ modelling the problem at hand
- ▶ designing objects to represent that model
- ▶ designing the relations between these objects.

Relations

Q What kind of relations have we seen ?

Relations

Q What kind of relations have we seen ?

- ▶ “is an element of” (instances)
- ▶ “is a subset of” (implementation of an interface, subclass of a class)

Relations

Q What kind of relations have we seen ?

- ▶ “is an element of” (instances)
- ▶ “is a subset of” (implementation of an interface, subclass of a class) but also
- ▶ “is part of the definition of” (properties, i.e. fields and get/set)
- ▶ “requires a” (construction parameters)
- ▶ “sends messages to”

Design Patterns

Design patterns are a number of typical relations, often met in Object-Oriented Programming.

Design Patterns

Design patterns are a number of typical relations, often met in Object-Oriented Programming.

- ▶ Creation patterns.
- ▶ Structural patterns.
- ▶ Behavioral patterns.

Design Patterns

Design patterns are a number of typical relations, often met in Object-Oriented Programming.

- ▶ Creation patterns.
- ▶ Structural patterns.
- ▶ Behavioral patterns.

Today's lecture is about

- ▶ these relations
- ▶ when they appear
- ▶ how to implement them.

Design Patterns

Design patterns are a number of typical relations, often met in Object-Oriented Programming.

- ▶ Creation patterns.
- ▶ Structural patterns.
- ▶ Behavioral patterns.

Today's lecture is about

- ▶ these relations
- ▶ when they appear
- ▶ how to implement them.

(except, sometimes, I'll use the inverse order)

Introduction

Creation Patterns

Factories

Builders

Factory methods

Singletons

Structural Patterns

Adapter

Composition

Decoration

RTTI

Behavioural design patterns

Commands

Observer

Iterating

MVC

Java and memory management

References

Garbage-collection

Conclusions

Factories

The problem Interfaces let you work with a class without knowing exactly how it is implemented. But you can't actually construct the class without knowing how it is implemented.

Factories

The problem Interfaces let you work with a class without knowing exactly how it is implemented. But you can't actually construct the class without knowing how it is implemented.

The problem To instantiate a class, you need to know exactly what class you instantiate. Sometimes, that's just too much knowledge.

Factories

The problem Interfaces let you work with a class without knowing exactly how it is implemented. But you can't actually construct the class without knowing how it is implemented.

The problem To instantiate a class, you need to know exactly what class you instantiate. Sometimes, that's just too much knowledge.

Q When ?

Factories

The problem Interfaces let you work with a class without knowing exactly how it is implemented. But you can't actually construct the class without knowing how it is implemented.

The problem To instantiate a class, you need to know exactly what class you instantiate. Sometimes, that's just too much knowledge.

Q When ?

A

- ▶ When your actual class isn't written yet.
- ▶ When you think that you might completely redesign the class.

Factories

The problem Interfaces let you work with a class without knowing exactly how it is implemented. But you can't actually construct the class without knowing how it is implemented.

The problem To instantiate a class, you need to know exactly what class you instantiate. Sometimes, that's just too much knowledge.

Q When ?

A

- ▶ When your actual class isn't written yet.
- ▶ When you think that you might completely redesign the class.
- ▶ When your "constructor" should actually be able to create different classes, depending on the circumstances.

Factories

The problem Interfaces let you work with a class without knowing exactly how it is implemented. But you can't actually construct the class without knowing how it is implemented.

The problem To instantiate a class, you need to know exactly what class you instantiate. Sometimes, that's just too much knowledge.

Q When ?

A

- ▶ When your actual class isn't written yet.
- ▶ When you think that you might completely redesign the class.
- ▶ When your "constructor" should actually be able to create different classes, depending on the circumstances.

Example Firefox's content display manager needs to use a different class for web pages, images or videos.

Factories

The problem Interfaces let you work with a class without knowing exactly how it is implemented. But you can't actually construct the class without knowing how it is implemented.

Q So, how do you handle that situation ?

Factories

The problem Interfaces let you work with a class without knowing exactly how it is implemented. But you can't actually construct the class without knowing how it is implemented.

Q So, how do you handle that situation ?

A Create a method returning an object with that interface.

Factories

The problem Interfaces let you work with a class without knowing exactly how it is implemented. But you can't actually construct the class without knowing how it is implemented.

Q So, how do you handle that situation ?

A Create a method returning an object with that interface.

```
/**
 * Create a display for a given url.
 *
 * @param contentType The content for this url, as specified by the server.
 * @param url The url where the actual content is actually stored.
 *
 * @return a ContentDisplay with the necessary knowledge
 * @throws NoDisplayAvailable if there is no plug-in to display this kind of content
 */
public ContentDisplay getDisplayForContent(String contentType, URL url) throws NoDisplayAvailable {
    if (contentType.equals("text/html"))
        return new WebDisplay(url);
    else if (contentType.equals("image/jpeg"))
        return new ImageDisplay(url);
    else if (contentType.equals("application/mpeg"))
        return new VideoDisplay(url);
    // ...
}
```

Design Pattern !

This trick isn't much, but it's a useful one.

Design Pattern !

This trick isn't much, but it's a useful one.
This is the *Abstract Factory* Design Pattern.

Design Pattern !

This trick isn't much, but it's a useful one.

This is the *Abstract Factory* Design Pattern.

In many libraries and languages, it's considered the “best manner” of creating an object. In OCaml, it's actually the *only* way of creating objects.

Builders

The problem Sometimes, you don't just need to build an object, you also need to *always* initialise it.

Builders

The problem Sometimes, you don't just need to build an object, you also need to *always* initialise it.

Q When ?

Builders

The problem Sometimes, you don't just need to build an object, you also need to *always* initialise it.

Q When ?

A

- ▶ When you're not the one doing the construction (often because it's hidden by an abstract factory).
- ▶ When your class might be redesigned and you wish to hide the initialisation process from the user.

Builders

The problem Sometimes, you don't just need to build an object, you also need to *always* initialise it.

Q When ?

A

- ▶ When you're not the one doing the construction (often because it's hidden by an abstract factory).
- ▶ When your class might be redesigned and you wish to hide the initialisation process from the user.

Example In Firefox, after having created the `ContentDisplay`, you need to initialise it by telling it what window should be used for display.

Builders !

The problem Interfaces let you work with a class without knowing exactly how it is implemented. But you can't actually construct the class without knowing how it is implemented.

Q So, how do you handle that situation ?

Builders !

The problem Interfaces let you work with a class without knowing exactly how it is implemented. But you can't actually construct the class without knowing how it is implemented.

Q So, how do you handle that situation ?

A Again, create a method returning an object with that interface.

Builders !

The problem Interfaces let you work with a class without knowing exactly how it is implemented. But you can't actually construct the class without knowing how it is implemented.

Q So, how do you handle that situation ?

A Again, create a method returning an object with that interface.

```
/**  
 * Create and initialise a display for a given url.  
 *  
 * @param contentType The content for this url, as specified by the server.  
 * @param url The url where the actual content is actually stored.  
 * @param frame The window containing this display.  
 *  
 * @return a ContentDisplay with the necessary knowledge  
 * @throws NoDisplayAvailable if there is no plug-in to display this kind of content  
 */  
public ContentDisplay buildDisplayForContent(String contentType, URL url, Frame frame) {  
    ContentDisplay display = this.getDisplayForContent(contentType, url);  
    display.setFrame(frame);  
    return display;  
}
```

Notes about builders

The difference between abstract factories and builders is small. Often, an abstract factory will also be a builder.

Notes about builders

The difference between abstract factories and builders is small. Often, an abstract factory will also be a builder.

In Firefox, abstract factories are created automatically, while builders need to be constructed manually.

Notes about builders

The difference between abstract factories and builders is small. Often, an abstract factory will also be a builder.

In Firefox, abstract factories are created automatically, while builders need to be constructed manually.

Ready for more design patterns ?

Factory method

That's just a fancy name for saying that you may

- ▶ subclass your factory class
- ▶ override your factory method to add new possibilities.

Factory method

That's just a fancy name for saying that you may

- ▶ subclass your factory class
- ▶ override your factory method to add new possibilities.

Q Can anyone remind me what overriding is all about ?

Factory method

That's just a fancy name for saying that you may

- ▶ subclass your factory class
- ▶ override your factory method to add new possibilities.

Q Can anyone remind me what overriding is all about ?

A Overriding is *replacing* a method of a class in a subclass.

Factory method

That's just a fancy name for saying that you may

- ▶ subclass your factory class
- ▶ override your factory method to add new possibilities.

Q Can anyone remind me what overriding is all about ?

A Overriding is *replacing* a method of a class in a subclass.

Now, on to really different design patterns.

Singleton pattern

The problem Sometimes, you wish to be sure that there is only one instance of a class in memory.

Singleton pattern

The problem Sometimes, you wish to be sure that there is only one instance of a class in memory.

Q When ?

Singleton pattern

The problem Sometimes, you wish to be sure that there is only one instance of a class in memory.

Q When ?

A When that instance needs to be shared between all components.

Singleton pattern

The problem Sometimes, you wish to be sure that there is only one instance of a class in memory.

Q When ?

A When that instance needs to be shared between all components.

Examples

- ▶ The main class of an application.
- ▶ The class with the preferences.
- ▶ The main window of an application.
- ▶ The print manager...

Singleton pattern

The problem Sometimes, you wish to be sure that there is only one instance of a class in memory.

Q So, how do you handle that situation ?

Singleton pattern

The problem Sometimes, you wish to be sure that there is only one instance of a class in memory.

Q So, how do you handle that situation ?

At least two solutions:

- ▶ make the constructor private and make everything else static
- ▶ make the constructor private, create one instance and return it through a factory.

Singleton pattern

The problem Sometimes, you wish to be sure that there is only one instance of a class in memory.

Q So, how do you handle that situation ?

At least two solutions:

- ▶ make the constructor private and make everything else static
- ▶ make the constructor private, create one instance and return it through a factory.

Q Advantages and disadvantages ?

Singleton pattern

The problem Sometimes, you wish to be sure that there is only one instance of a class in memory.

Q So, how do you handle that situation ?

At least two solutions:

- ▶ make the constructor private and make everything else static
- ▶ make the constructor private, create one instance and return it through a factory.

Q Advantages and disadvantages ?

The second solution is more flexible but heavier.

Singleton pattern

The problem Sometimes, you wish to be sure that there is only one instance of a class in memory.

Q So, how do you handle that situation ?

At least two solutions:

- ▶ make the constructor private and make everything else static
- ▶ make the constructor private, create one instance and return it through a factory.

Q Advantages and disadvantages ?

The second solution is more flexible but heavier.

Examples Java's System (first solution), Java's Toolkit (second solution).

Here comes the singleton

```
public class OperatingSystem
{
    private static final OperatingSystem os = new OperatingSystem(); //The only instance

    private OperatingSystem() //This class can't be instantiated
    {
    }

    public static OperatingSystem getTheOperatingSystem() //This is the only way of
    {
        return os;
    }
}
```

Here comes the singleton

```
public class OperatingSystem
{
    private static final OperatingSystem os = new OperatingSystem(); //The only instance

    private OperatingSystem() //This class can't be instantiated
    {
    }

    public static OperatingSystem getTheOperatingSystem() //This is the only way of
    {
        return os;
    }
}
```

It collides somewhat with static.

Here comes the singleton

```
public class OperatingSystem
{
    private static final OperatingSystem os = new OperatingSystem(); //The only instance

    private OperatingSystem() //This class can't be instantiated
    {
    }

    public static OperatingSystem getTheOperatingSystem() //This is the only way of
    {
        return os;
    }
}
```

It collides somewhat with static.

It's a common trick in most languages, including OCaml.

What we've seen so far

Q What have we seen so far ?

What we've seen so far

Q What have we seen so far ?

A Alternative manners of building an object – or hiding how it's built.

What we've seen so far

Q What have we seen so far ?

A Alternative manners of building an object – or hiding how it's built.

A A few tools for your Object-Oriented Programming toolkit.

What we've seen so far

Q What have we seen so far ?

A Alternative manners of building an object – or hiding how it's built.

A A few tools for your Object-Oriented Programming toolkit.

The most important part is not the patterns themselves – they're not very complicated. What's important is the vocabulary, as it lets you discuss common solutions with other programmers.

What we've seen so far

Q What have we seen so far ?

A Alternative manners of building an object – or hiding how it's built.

A A few tools for your Object-Oriented Programming toolkit.

The most important part is not the patterns themselves – they're not very complicated. What's important is the vocabulary, as it lets you discuss common solutions with other programmers.

Now, on to more design patterns !

Introduction

Creation Patterns

Factories

Builders

Factory methods

Singletons

Structural Patterns

Adapter

Composition

Decoration

RTTI

Behavioural design patterns

Commands

Observer

Iterating

MVC

Java and memory management

References

Garbage-collection

Conclusions

Structural Patterns

Structural Design Patterns are all about

- ▶ combining objects together
- ▶ changing (or pretending to change) the structure of objects to fit existing designs.

Adapting

The problem Sometimes, the object you have doesn't fit the interface it should have.

Adapting

The problem Sometimes, the object you have doesn't fit the interface it should have.

Q When ?

Adapting

The problem Sometimes, the object you have doesn't fit the interface it should have.

Q When ?

A

- ▶ You're combining two libraries with different sets of interfaces.
- ▶ You're redesigning your code but want to stay compatible with your documentation.

Adapting

The problem Sometimes, the object you have doesn't fit the interface it should have.

Q When ?

A

- ▶ You're combining two libraries with different sets of interfaces.
- ▶ You're redesigning your code but want to stay compatible with your documentation.

Example Last week's `SimpleList` does not match Java's official `Collection` interface.

Adapting

The problem Sometimes, the object you have doesn't fit the interface it should have.

Q So, how do you handle that situation ?

Adapting

The problem Sometimes, the object you have doesn't fit the interface it should have.

Q So, how do you handle that situation ?

A Wrap it inside another class actually fitting the interface.

Adapting

The problem Sometimes, the object you have doesn't fit the interface it should have.

Q So, how do you handle that situation ?

A Wrap it inside another class actually fitting the interface.

See Dr.Java.

Adapting

The problem Sometimes, the object you have doesn't fit the interface it should have.

Q So, how do you handle that situation ?

A Wrap it inside another class actually fitting the interface.

See Dr.Java.

That's the Adaptation Design Pattern (if you're adapting existing code to an interface), or Proxy Design Pattern (if you're keeping your old code and adding a compatibility layer).

Composition

The situation Some components are essentially built from smaller components.

Q Any examples ?

Composition

The situation Some components are essentially built from smaller components.

Q Any examples ?

Examples

- ▶ Directories are made of directories and files.
- ▶ Hash-table internal nodes are made of hash-table internal nodes and leaves.
- ▶ Swing components are made of primitive components and other components.

Composition

The situation Some components are essentially built from smaller components.

Q Any examples ?

Examples

- ▶ Directories are made of directories and files.
- ▶ Hash-table internal nodes are made of hash-table internal nodes and leaves.
- ▶ Swing components are made of primitive components and other components.
- ▶ Well, everything you represent with a tree.

Composition

The situation Some components are essentially built from smaller components.

Q Any examples ?

Examples

- ▶ Directories are made of directories and files.
- ▶ Hash-table internal nodes are made of hash-table internal nodes and leaves.
- ▶ Swing components are made of primitive components and other components.
- ▶ Well, everything you represent with a tree.

Q Any suggestion regarding how to best do it ?

Composition

The situation Some components are essentially built from smaller components.

Q Any examples ?

Examples

- ▶ Directories are made of directories and files.
- ▶ Hash-table internal nodes are made of hash-table internal nodes and leaves.
- ▶ Swing components are made of primitive components and other components.
- ▶ Well, everything you represent with a tree.

Q Any suggestion regarding how to best do it ?

A Use a common interface for leaves and internal nodes. See last week's Huffman tree.

Composition

The situation Some components are essentially built from smaller components.

Q Any examples ?

Examples

- ▶ Directories are made of directories and files.
- ▶ Hash-table internal nodes are made of hash-table internal nodes and leaves.
- ▶ Swing components are made of primitive components and other components.
- ▶ Well, everything you represent with a tree.

Q Any suggestion regarding how to best do it ?

A Use a common interface for leaves and internal nodes. See last week's Huffman tree.

That's the Composition Design Pattern.

Decoration

The problem Some objects need to have additional features added during the execution.

Decoration

The problem Some objects need to have additional features added during the execution.

Q When ?

Decoration

The problem Some objects need to have additional features added during the execution.

Q When ?

When there are many extensions you can add to a class and you don't want to add them all to all classes, as it's not practical or you don't know during the design phase which objects will need the classes.

Decoration

The problem Some objects need to have additional features added during the execution.

Q When ?

When there are many extensions you can add to a class and you don't want to add them all to all classes, as it's not practical or you don't know during the design phase which objects will need the classes.

Examples Dynamically adding scroll bars or shadows to a user-interface component.

Decoration

The problem Some objects need to have additional features added during the execution.

Decoration

The problem Some objects need to have additional features added during the execution.

Q So, how do you handle that situation ?

Decoration

The problem Some objects need to have additional features added during the execution.

Q So, how do you handle that situation ?

A Replace your object with a new object with the same interface, handling only the additional feature, and delegating everything else to the old object.

Decoration

The problem Some objects need to have additional features added during the execution.

Q So, how do you handle that situation ?

A Replace your object with a new object with the same interface, handling only the additional feature, and delegating everything else to the old object.

See Dr. Java.

Tired about design patterns ?

Q Do you have enough Design Patterns ?

Tired about design patterns ?

Q Do you have enough Design Patterns ?

A Well, I do. So let's move to another subject for the moment.

Introduction

Creation Patterns

- Factories

- Builders

- Factory methods

- Singletons

Structural Patterns

- Adapter

- Composition

- Decoration

RTTI

Behavioural design patterns

- Commands

- Observer

- Iterating

- MVC

Java and memory management

- References

- Garbage-collection

Conclusions

Run-time Type Information

As you have already seen, in Java, classes are objects themselves: if `MyClass` is a class, then `MyClass.class` is an object of class `Class`, with all the methods of class `Class`.

Run-time Type Information

As you have already seen, in Java, classes are objects themselves: if `MyClass` is a class, then `MyClass.class` is an object of class `Class`, with all the methods of class `Class`.

This goes further.

Going further

If you have an object `myObject`, then

- ▶ `myObject.getClass()` will tell you the actual class of this object.
- ▶ `(MyOtherClass)myObject` will attempt to consider this object as a member of another class – that's called *casting*
- ▶ `myObject instanceof MyClass` will return `true` if this object is a member of that other class.

Going further

If you have an object `myObject`, then

- ▶ `myObject.getClass()` will tell you the actual class of this object.
- ▶ `(MyOtherClass)myObject` will attempt to consider this object as a member of another class – that's called *casting*
- ▶ `myObject instanceof MyClass` will return `true` if this object is a member of that other class.

Let's look at Dr.Java.

Going further

If you have an object `myObject`, then

- ▶ `myObject.getClass()` will tell you the actual class of this object.
- ▶ `(MyOtherClass)myObject` will attempt to consider this object as a member of another class – that's called *casting*
- ▶ `myObject instanceof MyClass` will return `true` if this object is a member of that other class.

Let's look at Dr.Java.

Be careful with *casting*, as it causes errors when used improperly.

Number conversion

Casting has an additional application: number conversion.

Number conversion

Casting has an additional application: number conversion.
Example in Dr.Java.

Conclusions

Java lets you

- ▶ find out about type information during the execution of a program
- ▶ mess with that type information !

Conclusions

Java lets you

- ▶ find out about type information during the execution of a program
- ▶ mess with that type information !

Among other things, this means that Java keeps type information during the execution of the program. That can be useful, but also memory consuming.

Conclusions

Java lets you

- ▶ find out about type information during the execution of a program
- ▶ mess with that type information !

Among other things, this means that Java keeps type information during the execution of the program. That can be useful, but also memory consuming.

Roughly half of the programming languages do that. Java, Python, Ruby, C#, C++... keep some or all type information, while OCaml, Haskell, C... don.◻

Introduction

Creation Patterns

Factories

Builders

Factory methods

Singletons

Structural Patterns

Adapter

Composition

Decoration

RTTI

Behavioural design patterns

Commands

Observer

Iterating

MVC

Java and memory management

References

Garbage-collection

Conclusions

Complex method calls

The problem Sometimes, a method call is not (directly) the right way of representing a message.

Q When ?

Complex method calls

The problem Sometimes, a method call is not (directly) the right way of representing a message.

Q When ?

A When the message is too complex (say, 20 fields), too confusing (say, all fields are integers), or when you might need to add information after having designed the protocol, or when the message will need to be routed to its correct destination, or when the message will need to wait until it can be treated.

Complex method calls

The problem Sometimes, a method call is not (directly) the right way of representing a message.

Q When ?

A When the message is too complex (say, 20 fields), too confusing (say, all fields are integers), or when you might need to add information after having designed the protocol, or when the message will need to be routed to its correct destination, or when the message will need to wait until it can be treated.

Example User Interface events – most components don't need to know all the details about mouse movements or time events. When you plug a 20-buttons-mouse on the computer, you don't want to change the methods to be able to accept 20 parameters.

Command

The problem Sometimes, a method call is not (directly) the right way of representing a message.

Q So, how do you handle that situation ?

Command

The problem Sometimes, a method call is not (directly) the right way of representing a message.

Q So, how do you handle that situation ?

Instead of a message with 20 fields, use a message with 1 field – and make that field a (potentially complex) object.

Command

The problem Sometimes, a method call is not (directly) the right way of representing a message.

Q So, how do you handle that situation ?

Instead of a message with 20 fields, use a message with 1 field – and make that field a (potentially complex) object.

Example `actionListeners`

Command design pattern

This is the *command* design pattern:

- ▶ omnipresent in UI design (for events)

Command design pattern

This is the *command* design pattern:

- ▶ omnipresent in UI design (for events)
- ▶ omnipresent in concurrency (also for events)
- ▶ used in applications (to make Undos easier)
- ▶ used in applications (as macros)
- ▶ used in network applications (as complex messages)
- ▶ etc.

Observation

The problem Observing changes to the state of an object.
Q When ?

Observation

The problem Observing changes to the state of an object.

Q When ?

A When you need to react to these changes, to update the state of your own object, for instance to show a coherent view to the user.

Examples Again, User Interfaces (buttons changing depending where you're in the text, but also components reacting to mouse clicks, animations reacting to time...), but also games, operating systems...

Observation

The problem Observing changes to the state of an object.

Q So, how do you handle that situation ?

Observation

The problem Observing changes to the state of an object.

Q So, how do you handle that situation ?

A Just as Java handles events. With:

- ▶ an interface for objects that might be interested in the event (say `actionListener`)
- ▶ a subscription mechanism (say `addActionListener`)
- ▶ some data structure inside the observed object to record all listeners
- ▶ every method causing a change must trigger the emission of a message to all registered listeners.

Observation

The problem Observing changes to the state of an object.

Q So, how do you handle that situation ?

A Just as Java handles events. With:

- ▶ an interface for objects that might be interested in the event (say `actionListener`)
- ▶ a subscription mechanism (say `addActionListener`)
- ▶ some data structure inside the observed object to record all listeners
- ▶ every method causing a change must trigger the emission of a message to all registered listeners.
that's the difficult part, as it's quite tricky to retrofit

Observation

The problem Observing changes to the state of an object.

Q So, how do you handle that situation ?

A Just as Java handles events. With:

- ▶ an interface for objects that might be interested in the event (say `actionListener`)
- ▶ a subscription mechanism (say `addActionListener`)
- ▶ some data structure inside the observed object to record all listeners
- ▶ every method causing a change must trigger the emission of a message to all registered listeners.
that's the difficult part, as it's quite tricky to retrofit
that's why it's a good idea to have setter methods !

Observer

This is the *Observer* pattern, also called *Publish/Subscribe*.

Observer

This is the *Observer* pattern, also called *Publish/Subscribe*.

Some languages don't need this pattern, as it's built-in (Hypercard, old versions of Visual Basic, Trolltech's extended C++) or fully replaced by message passing (Erlang, JoCaml...)

Iteration

The problem Often, you have a (complex) data structure, whose internals you do not wish to show. Still, other programmers will need to use the data from your data structure.

Q When ?

Iteration

The problem Often, you have a (complex) data structure, whose internals you do not wish to show. Still, other programmers will need to use the data from your data structure.

Q When ?

A When the internals of your data structure may change, be secret, or just be horribly complex. Which is, basically, all the time.

Iteration

The problem Often, you have a (complex) data structure, whose internals you do not wish to show. Still, other programmers will need to use the data from your data structure.

Q When ?

A When the internals of your data structure may change, be secret, or just be horribly complex. Which is, basically, all the time.

Examples Just about every single data structure, including hash tables, arrays, linked lists. . .

Iteration

The problem Often, you have a (complex) data structure, whose internals you do not wish to show. Still, other programmers will need to use the data from your data structure.

Q So, how do you handle that situation ?

Iteration

The problem Often, you have a (complex) data structure, whose internals you do not wish to show. Still, other programmers will need to use the data from your data structure.

Q So, how do you handle that situation ?

A At least two possibilities:

- ▶ give a standard way to enumerate all elements of your data structure (Java-style) – using iterators
- ▶ make your data structure accept a task and apply it to everyone in the data structure (OCaml-style) – using λ -abstractions/delegates

Iteration

The problem Often, you have a (complex) data structure, whose internals you do not wish to show. Still, other programmers will need to use the data from your data structure.

Q So, how do you handle that situation ?

A At least two possibilities:

- ▶ give a standard way to enumerate all elements of your data structure (Java-style) – using iterators
- ▶ make your data structure accept a task and apply it to everyone in the data structure (OCaml-style) – using λ -abstractions/delegates

Q Which one is best ?

Iteration

The problem Often, you have a (complex) data structure, whose internals you do not wish to show. Still, other programmers will need to use the data from your data structure.

Q So, how do you handle that situation ?

A At least two possibilities:

- ▶ give a standard way to enumerate all elements of your data structure (Java-style) – using iterators
- ▶ make your data structure accept a task and apply it to everyone in the data structure (OCaml-style) – using λ -abstractions/delegates

Q Which one is best ?

A That's a matter of religion. In OCaml, the iterator-style is somewhat complex and the λ -abstraction-style is trivial. In Java, the iterator-style is somewhat complex and the λ -abstraction-style is way too complex.

Iteration

The problem Often, you have a (complex) data structure, whose internals you do not wish to show. Still, other programmers will need to use the data from your data structure.

Q So, how do you handle that situation ?

A At least two possibilities:

- ▶ give a standard way to enumerate all elements of your data structure (Java-style) – using iterators
- ▶ make your data structure accept a task and apply it to everyone in the data structure (OCaml-style) – using λ -abstractions/delegates

Q Which one is best ?

A That's a matter of religion. In OCaml, the iterator-style is somewhat complex and the λ -abstraction-style is trivial. In Java, the iterator-style is somewhat complex and the λ -abstraction-style is way too complex.

Examples see the examples of the previous weeks, including file navigation, etc.

Model-View-Controller

This one is the most important one. It's probably not a design pattern, but it's still the most important one.

Model-View-Controller

This one is the most important one. It's probably not a design pattern, but it's still the most important one. Or at least the biggest.

Model-View-Controller

This one is the most important one. It's probably not a design pattern, but it's still the most important one. Or at least the biggest.

The question Quite often, when you design an application with a GUI, you want to make sure that your GUI and the rest of the application are kept logically separate.

Model-View-Controller

This one is the most important one. It's probably not a design pattern, but it's still the most important one. Or at least the biggest.

The question Quite often, when you design an application with a GUI, you want to make sure that your GUI and the rest of the application are kept logically separate.

Q Why ?

Model-View-Controller

This one is the most important one. It's probably not a design pattern, but it's still the most important one. Or at least the biggest.

The question Quite often, when you design an application with a GUI, you want to make sure that your GUI and the rest of the application are kept logically separate.

Q Why ?

A

- ▶ you should be able to fix bugs in the internals without having to change the UI
- ▶ you should be able to fix bugs in the UI without having to change the internals
- ▶ the user interface and the internals are typically not designed (or implemented) by the same team
- ▶ you can have several different User Interfaces for the same internals (think BitTorrent)
- ▶ the same User Interface can drive different internals (think Windows 2000 vs Windows 98)
- ▶ ...

Model-View-Controller

This one is the most important one. It's probably not a design pattern, but it's still the most important one. Or at least the biggest.

The question Quite often, when you design an application with a GUI, you want to make sure that your GUI and the rest of the application are kept logically separate.

Q Why ?

A

- ▶ you should be able to fix bugs in the internals without having to change the UI
- ▶ you should be able to fix bugs in the UI without having to change the internals
- ▶ the user interface and the internals are typically not designed (or implemented) by the same team
- ▶ you can have several different User Interfaces for the same internals (think BitTorrent)
- ▶ the same User Interface can drive different internals (think Windows 2000 vs Windows 98)
- ▶ ...

Model-View-Controller

The question Quite often, when you design an application with a GUI, you want to make sure that your GUI and the rest of the application are kept logically separate.

Q So, how do you handle that situation ?

Model-View-Controller

The question Quite often, when you design an application with a GUI, you want to make sure that your GUI and the rest of the application are kept logically separate.

Q So, how do you handle that situation ?

The usual answer is to separate

- model** the domain logic (e.g. the rules of the game, the content of your documents, etc.)
- controller** the tools given to the user to control the application (e.g. toolbars, menus, buttons, etc.)
- view** the visual feedback (e.g. the grid, the icons, the webpage, etc.)

Model-View-Controller

The question Quite often, when you design an application with a GUI, you want to make sure that your GUI and the rest of the application are kept logically separate.

Q So, how do you handle that situation ?

The usual answer is to separate

model the domain logic (e.g. the rules of the game, the content of your documents, etc.)

controller the tools given to the user to control the application (e.g. toolbars, menus, buttons, etc.)

view the visual feedback (e.g. the grid, the icons, the webpage, etc.)

Typically, all three components communicate by messages:

- ▶ the controller informs the models that something has happened and that it should change
- ▶ the controller informs the view that something has changed and that the feedback should be updated

Model-View-Controller

The question Quite often, when you design an application with a GUI, you want to make sure that your GUI and the rest of the application are kept logically separate.

Q So, how do you handle that situation ?

The usual answer is to separate

- model** the domain logic (e.g. the rules of the game, the content of your documents, etc.)
- controller** the tools given to the user to control the application (e.g. toolbars, menus, buttons, etc.)
- view** the visual feedback (e.g. the grid, the icons, the webpage, etc.)

Typically, all three components communicate by messages:

- ▶ the controller informs the models that something has happened and that it should change
- ▶ the controller informs the view that something has changed and that the feedback should be updated

Note that you can typically plug several different controllers and/or view to the same model. The contrary is usually harder.

Model-View-Controller

The question Quite often, when you design an application with a GUI, you want to make sure that your GUI and the rest of the application are kept logically separate.

Q So, how do you handle that situation ?

The usual answer is to separate

- model** the domain logic (e.g. the rules of the game, the content of your documents, etc.)
- controller** the tools given to the user to control the application (e.g. toolbars, menus, buttons, etc.)
- view** the visual feedback (e.g. the grid, the icons, the webpage, etc.)

Typically, all three components communicate by messages:

- ▶ the controller informs the models that something has happened and that it should change
- ▶ the controller informs the view that something has changed and that the feedback should be updated

Note that you can typically plug several different controllers and/or view to the same model. The contrary is usually harder.

Enough with Design Patterns

By now, you're probably fed up with design patterns. So am I.

Enough with Design Patterns

By now, you're probably fed up with design patterns. So am I.
Still, keep them somewhere in a corner of your mind. They might come in handy at some point.

Introduction

Creation Patterns

- Factories

- Builders

- Factory methods

- Singletons

Structural Patterns

- Adapter

- Composition

- Decoration

RTTI

Behavioural design patterns

- Commands

- Observer

- Iterating

- MVC

Java and memory management

- References

- Garbage-collection

Conclusions

Memory

Memory is a limited resource. It can be
allocated to an object
used
referenced
deallocated .

Memory

Memory is a limited resource. It can be

allocated to an object When ?

used

referenced How ?

deallocated .How ?

Memory

Memory is a limited resource. It can be

allocated to an object `new`

used

referenced How ?

deallocated .How ?

,

Memory

Memory is a limited resource. It can be
allocated to an object new
used
referenced with a variable
deallocated .How ?

Memory

Memory is a limited resource. It can be

`allocated` to an object `new`

`used`

`referenced` with a variable

`deallocated` . good question - let's talk about that later

References

Warning Advanced Subject !

References

Warning Advanced Subject !

```
public static final void f(Object o)
{
    // ...
}
// ...
Object a = // ...
f(a);
```

References

Warning Advanced Subject !

```
public static final void f(Object o)
{
    // ...
}
// ...
Object a = // ...
f(a);
```

What exactly is going on during the invocation of f ?

References

Warning Advanced Subject !

```
public static final void f(Object o)
{
    // ...
}
// ...
Object a = // ...
f(a);
```

What exactly is going on during the invocation of f ?

- ▶ is o identical to a ?
- ▶ is o a copy of a ?
- ▶ is there a difference between o and a ?

References

- ▶ is o identical to a ? – no – can you prove it ?
- ▶ is o a copy of a ? – the object is not copied – can you prove it ?
- ▶ is there a difference between o and a ? – well, yes

References

- ▶ is o identical to a ? – no – can you prove it ? think $o \leftarrow \text{null}$
- ▶ is o a copy of a ? – the object is not copied – can you prove it ?
- ▶ is there a difference between o and a ? – well, yes

References

- ▶ is o identical to a ? – no – can you prove it? think $o \leftarrow null$
- ▶ is o a copy of a ? – the object is not copied – can you prove it? think arrays
- ▶ is there a difference between o and a ? – well, yes

The truth is that a is *not* actually your object. Your object is somewhere in memory. What a is is a *reference* to your object, that is the actual address in memory where your object is stored.

References

- ▶ is o identical to a ? – no – can you prove it? think $o \leftarrow null$
- ▶ is o a copy of a ? – the object is not copied – can you prove it? think arrays
- ▶ is there a difference between o and a ? – well, yes

The truth is that a is *not* actually your object. Your object is somewhere in memory. What a is is a *reference* to your object, that is the actual address in memory where your object is stored.

References

- ▶ is o identical to a ? – no – can you prove it ? think $o \leftarrow null$
- ▶ is o a copy of a ? – the object is not copied – can you prove it ? think arrays
- ▶ is there a difference between o and a ? – well, yes

The truth is that a is *not* actually your object. Your object is somewhere in memory. What a is is a *reference* to your object, that is the actual address in memory where your object is stored.

During the invocation, a is copied into o .

- ▶ it's a fast operation
- ▶ it permits sharing data structures
- ▶ it's simple.

Remember `null` ?

Since o and a are only addresses – and not objects themselves – they can have values that do not really represent an object. Say random addresses.

Remember `null` ?

Since o and a are only addresses – and not objects themselves – they can have values that do not really represent an object. Say random addresses. In C and C++, that's the biggest source of problems, as *pointers* can point anywhere inside RAM.

Remember `null` ?

Since `o` and `a` are only addresses – and not objects themselves – they can have values that do not really represent an object. Say random addresses. In C and C++, that's the biggest source of problems, as *pointers* can point anywhere inside RAM. Or outside it, actually.

Remember `null` ?

Since `o` and `a` are only addresses – and not objects themselves – they can have values that do not really represent an object. Say random addresses. In C and C++, that's the biggest source of problems, as *pointers* can point anywhere inside RAM. Or outside it, actually.

Anyway, in Java, a reference always contains the address of

- ▶ an existing object; or
- ▶ `null` – that is, address 0, meaning “no object”.

Nullable and non-nullable

If you remember, some types cannot receive `null`:

- ▶ `boolean`
- ▶ `byte` (numbers between -128 and +127)
- ▶ `character`
- ▶ `double`
- ▶ `float`
- ▶ `int`
- ▶ `short`
- ▶ `void`

Nullable and non-nullable

If you remember, some types cannot receive `null`:

- ▶ `boolean`
- ▶ `byte` (numbers between -128 and +127)
- ▶ `character`
- ▶ `double`
- ▶ `float`
- ▶ `int`
- ▶ `short`
- ▶ `void`

These are the so-called “primitive types”: they are understood directly by the (virtual) machine. They also cannot be built or referenced.

Consequently, they also cannot be *dereferenced*. They are the *only* values without fields or methods.

Nullable and non-nullable

If you remember, some types cannot receive `null`:

- ▶ `boolean`
- ▶ `byte` (numbers between -128 and +127)
- ▶ `character`
- ▶ `double`
- ▶ `float`
- ▶ `int`
- ▶ `short`
- ▶ `void`

These are the so-called “primitive types”: they are understood directly by the (virtual) machine. They also cannot be built or referenced.

Consequently, they also cannot be *dereferenced*. They are the *only* values without fields or methods.

That’s because these types are extremely short. It’s much more efficient in terms of memory to store them directly (without an address) and to copy them around.

Nullable and non-nullable

If you remember, some types cannot receive `null`:

- ▶ `boolean`
- ▶ `byte` (numbers between -128 and +127)
- ▶ `character`
- ▶ `double`
- ▶ `float`
- ▶ `int`
- ▶ `short`
- ▶ `void`

These are the so-called “primitive types”: they are understood directly by the (virtual) machine. They also cannot be built or referenced.

Consequently, they also cannot be *dereferenced*. They are the *only* values without fields or methods.

That’s because these types are extremely short. It’s much more efficient in terms of memory to store them directly (without an address) and to copy them around.

In fact, there is one more primitive type – it just happens that this type does not have a Java name. It’s the type of references themselves.

It's not that simple.

For each of these native types, there's a *wrapper* (or *boxed*) type, allowing (mostly) automatic conversion between a primitive type and an object.

It's not that simple.

For each of these native types, there's a *wrapper* (or *boxed*) type, allowing (mostly) automatic conversion between a primitive type and an object.

- ▶ `boolean` – `Boolean`
- ▶ `byte` – `Byte`
- ▶ `char` – `Character`
- ▶ `double` – `Double`
- ▶ `float` – `Float`
- ▶ `int` – `Integer`
- ▶ `short` – `Short`
- ▶ `void` – `Void`

It's not that simple.

For each of these native types, there's a *wrapper* (or *boxed*) type, allowing (mostly) automatic conversion between a primitive type and an object.

- ▶ `boolean` – `Boolean`
- ▶ `byte` – `Byte`
- ▶ `char` – `Character`
- ▶ `double` – `Double`
- ▶ `float` – `Float`
- ▶ `int` – `Integer`
- ▶ `short` – `Short`
- ▶ `void` – `Void` – that one is not used very often

It's not that simple.

For each of these native types, there's a *wrapper* (or *boxed*) type, allowing (mostly) automatic conversion between a primitive type and an object.

- ▶ `boolean` – `Boolean`
- ▶ `byte` – `Byte`
- ▶ `char` – `Character`
- ▶ `double` – `Double`
- ▶ `float` – `Float`
- ▶ `int` – `Integer`
- ▶ `short` – `Short`
- ▶ `void` – `Void` – that one is not used very often

Conversion between the primitive type and the corresponding class is generally automatic – but it's also a slow operation.

Misbehaviours

In Java, we have `1 == 1` but `"1" != "1"`. We also have `new Integer(1) != new Integer(1)` but, if we define `Integer a = 1;` `Integer b = 1;`, we have `a==b`. More surprisingly, if we define `Integer a = 15000;` `Integer b = 15000;`, we have `a!=b` !

Misbehaviours

In Java, we have `1 == 1` but `"1" != "1"`. We also have `new Integer(1) != new Integer(1)` but, if we define `Integer a = 1;`
`Integer b = 1;`, we have `a==b`. More surprisingly, if we define `Integer a = 15000;`
`Integer b = 15000;`, we have `a!=b` !

Q Why ?

Misbehaviours

In Java, we have `1 == 1` but `"1" != "1"`. We also have `new Integer(1) != new Integer(1)` but, if we define `Integer a = 1;`
`Integer b = 1;`, we have `a==b`. More surprisingly, if we define `Integer a = 15000;`
`Integer b = 15000;`, we have `a!=b` !

Q Why ?

A Because `==` compares only primitive types. In other words, it compares the references, not the contents of objects.

Misbehaviours

In Java, we have `1 == 1` but `“1” != “1”`. We also have `new Integer(1) != new Integer(1)` but, if we define `Integer a = 1;`
`Integer b = 1;`, we have `a==b`. More surprisingly, if we define `Integer a = 15000;`
`Integer b = 15000;`, we have `a!=b` !

Q Why ?

A Because `==` compares only primitive types. In other words, it compares the references, not the contents of objects.

Oh, and there's a special `Integer` for 0 and a special `Integer` for 1.
That's a hack.

Misbehaviours

In Java, we have `1 == 1` but `“1” != “1”`. We also have `new Integer(1) != new Integer(1)` but, if we define `Integer a = 1;` `Integer b = 1;`, we have `a==b`. More surprisingly, if we define `Integer a = 15000;` `Integer b = 15000;`, we have `a!=b` !

Q Why ?

A Because `==` compares only primitive types. In other words, it compares the references, not the contents of objects.

Oh, and there's a special `Integer` for 0 and a special `Integer` for 1.

That's a hack.

Note that this problem doesn't appear in OCaml.

Garbage-collection

Just a quick word about memory management.

Garbage-collection

Just a quick word about memory management.

Allocation is the easy part: you tell the program when to allocate memory.

Garbage-collection

Just a quick word about memory management.

Allocation is the easy part: you tell the program when to allocate memory.

Deallocation harder. When and how should Java destroy an object ?

Memory deallocation

Memory deallocation

Manual In C, C++, Pascal. . . The programmer must tell the program each time an object has to be destroyed. Objects are (almost) never destroyed automatically.

Memory deallocation

Manual In C, C++, Pascal. . . The programmer must tell the program each time an object has to be destroyed. Objects are (almost) never destroyed automatically.

Reference-counting In Python, Visual Basic, JavaScript 1. . . The program automatically detects when noone has any reference to a given object and it removes the corresponding object.

Memory analysis In Java, C#, OCaml. . . Every so often, a bit of the program wakes up and destroys objects, when there is no way the object can be used anymore.

Static analysis In MLKit, Cyclone. . . The compiler proves mathematically when objects are not used anymore and automatically inserts instructions to destroy the objects.

Memory deallocation

Manual In C, C++, Pascal. . . The programmer must tell the program each time an object has to be destroyed. Objects are (almost) never destroyed automatically.

Reference-counting In Python, Visual Basic, JavaScript 1. . . The program automatically detects when noone has any reference to a given object and it removes the corresponding object.

Memory analysis In Java, C#, OCaml. . . Every so often, a bit of the program wakes up and destroys objects, when there is no way the object can be used anymore.

Static analysis In MLKit, Cyclone. . . The compiler proves mathematically when objects are not used anymore and automatically inserts instructions to destroy the objects.

The things to remember is that

- ▶ in Java, memory management is automatic. . .
- ▶ but it can be tricked into leaking (Swing certainly does).

Introduction

Creation Patterns

Factories

Builders

Factory methods

Singletons

Structural Patterns

Adapter

Composition

Decoration

RTTI

Behavioural design patterns

Commands

Observer

Iterating

MVC

Java and memory management

References

Garbage-collection

Conclusions

What we've seen recently

1. vocabulary
2. common solutions to common problems
3. a few additional features of Java
4. and a few hints on how Java operates.

Introduction

Creation Patterns

- Factories

- Builders

- Factory methods

- Singletons

Structural Patterns

- Adapter

- Composition

- Decoration

RTTI

Behavioural design patterns

- Commands

- Observer

- Iterating

- MVC

Java and memory management

- References

- Garbage-collection

Conclusions

A few words

- ▶ Don't forget to work during your revision week.
- ▶ Ask questions while you can !

Now, questions