



Usando o Voyager

■ Voyager

- mais que uma simples plataforma para agentes móveis
- ORB completo
- versão atual = 3.1

■ Instalação

- requer JDK instalado e configurado
- configurações adicionais
 - l variável de ambiente CLASSPATH deve apontar para a localização do archive "voyager.jar", normalmente em \<voyager_path>\lib

■ Execução do ORB

- programa voyager.exe
- sintaxe: "voyager portnumber"



Usando o Voyager

■ Inicialização do ORB

- manual: usando o programa "voyager"
- automática: a partir de um programa Java

■ Classe "Voyager"

- contém métodos estáticos para iniciar, encerrar e monitorar o ORB Voyager (também chamado de "Voyager system")
- Não é possível criar um objeto desta classe

■ Principais Métodos

- `static void shutdown()`
- `static void startup()` - cliente: não aceita mensagens
- `static void startup(java.lang.Object object, java.lang.String url)`
 - servidor: usado para applets e servlets
- `static void startup(java.lang.String url)` - servidor



Usando Interfaces

■ Voyager

- usa massivamente o conceito de interface do Java
- Se MinhaClasse é uma classe do Voyager
 - IMinhaClasse é a interface que MinhaClasse implementa

■ Idéia

- tanto o objeto original quanto o proxy de um objeto remoto implementam uma mesma interface
- variáveis correspondentes a objetos remotos ou que serão exportados são interfaces
 - `IMinhaClasse mc = new MinhaClasse();`

■ Objetos

- podem ser criados localmente ou remotamente
- Voyager propicia mecanismo padrão para criação de objetos



Fabricando Objetos

■ Classe "Factory"

- contém métodos estáticos para a construção de objetos remotos

■ Principais Métodos

- `static Proxy create(java.lang.String classname)`
- `static Proxy create(java.lang.String classname, java.lang.Object[] args)`
- `static Proxy create(java.lang.String classname, java.lang.Object[] args, java.lang.String url)`
- `static Proxy create(java.lang.String classname, java.lang.String url)`
- `static Proxy create(java.lang.String classname, java.lang.String signature, java.lang.Object[] args, java.lang.String name)`
 - ┆ Retornam um proxy para uma nova instância da classe especificada



Fabricando Objetos

■ Classe "Proxy"

- Um proxy representa um outro objeto
- Um proxy implementa a mesma interface que este objeto implementa
- Mensagens enviadas ao objeto proxy são retransmitidas ao objeto que representam, mesmo que este esteja em uma máquina virtual diferente

■ Métodos Estáticos

- `static Proxy export(java.lang.Object object)`
- `static Proxy export(java.lang.Object object, java.lang.String url)`
 - | exportam "object"
- `static Proxy of(java.lang.Object object)`
- `static Proxy of(java.lang.Object object, java.lang.Class[] interfaces)` - retornam um proxy para "object"



Serviço de Nomes

■ Classe "Namespace"

- contém métodos estáticos que permitem a associação de identificadores a objetos para consulta futura
- identificadores podem ser simples "strings", um URL indicando outro servidor Voyager, um "path" de diretório ou um formato reconhecido por um dos layers de transporte plugáveis do Voyager (por exemplo o CORBA IOR)
- uma vez associado, um identificador pode ser desassociado e associado de novo

■ Métodos Principais

- `static void bind(java.lang.String name, java.lang.Object object)`
 - | Associa "name" a "object"
- `static java.lang.Object lookup(java.lang.String name)`
 - | Retorna um proxy para o objeto associado a "name"



Exemplo

■ Programa BasicoA.java

```
public class BasicoA
{
    public static void main( String[] args )
    {try
        {
            Voyager.startup();
            IStockmarket market = (IStockmarket) Proxy.export( new
                Stockmarket(), "9000" );
            Namespace.bind( "9000/NASDAQ", market );
        }
    catch( Exception exception )
    {
        System.err.println( exception );
    }
}
```



Exemplo

■ Programa BasicoB.java

```
public class BasicoB
{
    public static void main( String[] args )
    {
        try
        {
            Voyager.startup();
            IStockmarket market = (IStockmarket) Namespace.lookup(
                "//localhost:9000/NASDAQ" );
            market.news( "Sun releases Java" );
        }
        catch( Exception exception )
        {
            System.err.println( exception );
        }
    }
}
```




Agregação Dinâmica

■ Agregação Dinâmica

- permite a adição de código e dados a um objeto em tempo de execução
- representa um passo adiante para a modelagem de objetos, complementando os mecanismos de herança e polimorfismo

■ Resolve os seguintes problemas

- adicionar um novo comportamento a um componente legado, cuja fonte não se encontra disponível
- customização de um objeto considerando-se requisitos específicos a um subsistema, de tal forma que ele pode ser usado por múltiplos sistemas
- estender o comportamento de um objeto em tempo de execução, talvez de maneiras imprevisíveis



Agregação Dinâmica

■ Facet

- objetos secundários agregados a objetos primários, em tempo de execução

■ Objeto primário e seus facets

- formam um agregado que é tipicamente armazenado persistentemente, movido e processado pelo mecanismo de garbage-collection como uma entidade única

■ Características de um Facet

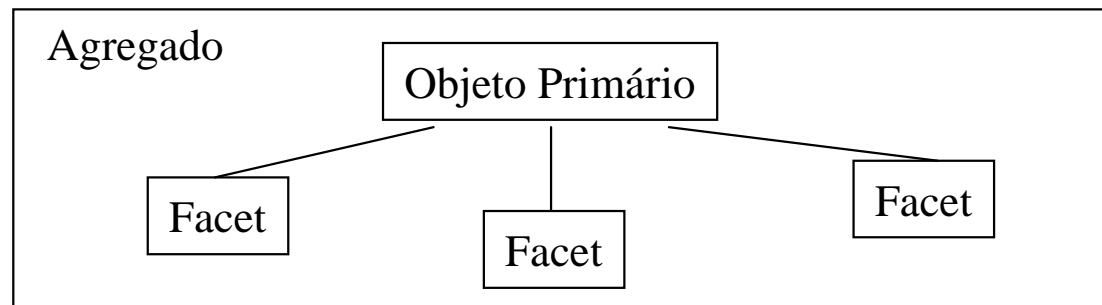
- Uma classe não precisa ser modificada para que uma instância sua seja um objeto primário ou um facet
- A classe de um facet não precisa estar relacionada de nenhuma maneira à classe de um objeto primário. Uma instância de uma classe pode ser adicionada como um facet a qualquer tipo de objeto primário



Agregação Dinâmica

■ Restrições a um Facet

- Facets não podem ser aninhados. Em outras palavras, um facet não pode ter seu próprio facet
- Facets não podem ser removidos. Uma vez criado, ele permanece por toda a vida útil do agregado como um todo
- Um objeto primário e seus facets tem a mesma vida útil que todos seus componentes individualmente. Eles são processados pelo mecanismo de garbage-collection somente quando não houverem mais referências nem ao objeto primário nem a nenhum de seus facets





Aggregação Dinâmica

■ Classe Facets

- Facets é uma implementação distribuída da interface Ifacets, que é a interface que permite o acesso aos facets de um objeto.
- Uma instância de Facets é automaticamente alocada a um objeto quando o primeiro facet é adicionado a ele.
- Esta instância é responsável por administrar os facets de um objeto e garantir que o objeto e seus facets vão ser processados pelo garbage collection como uma entidade única

■ Interface IFacets

- `java.lang.Object get(java.lang.String classname)`
- `java.lang.Object[] getFacets()`
- `java.lang.Object getPrimary()`
- `java.lang.Object of(java.lang.String classname)`



Aggregação Dinâmica

■ Classe Facets - Métodos Estáticos

- | `static IFacets get(java.lang.Object object)`
- | `static java.lang.Object get(java.lang.Object object, java.lang.Class type)`
 - | obtém facet para o objeto ***object***, se ele existe
- | `static IFacets of(java.lang.Object object)`
- | `static java.lang.Object of(java.lang.Object object, java.lang.Class type)`
 - | obtém Facet para o objeto ***object***, se ele já o possui, ou cria um se ele não existe

■ Classe Facets - Métodos que implementam IFacets

- | `java.lang.Object get(java.lang.String classname)`
- | `java.lang.Object[] getFacets()`
- | `java.lang.Object getPrimary()`
- | `java.lang.Object of(java.lang.String classname)`



Sistema de Mensagens

- Mensagens Síncronas
 - podem ser enviadas por meio da sintaxe regular do Java
- Outros tipos de mensagens
 - podem aumentar a flexibilidade do sistema
- Voyager
 - provê uma abstração para o envio de mensagens que suporta tipos de mensagens mais sofisticados
 - classes Sync, OneWay, Future permitem abstrair o mecanismo de mensagens, constituindo o sistema de mensagens do Voyager
- OneWay - mensagem assíncrona sem resposta
- Future - mensagem assíncrona com resposta



Sistema de Mensagens

■ Invocando Mensagens

- utilize o método `invoke`, disponível em `Sync`, `OneWay` e `Future`
- retorna um objeto da classe `Result`

■ Classe `Result`

- contém o resultado da mensagem, para os casos `Sync` e `Future`
- no caso de `Future`, pode ser inquirida via `polling` se a o retorno da mensagem já está disponível com o método `isAvailable()`.

■ Multicast

- O `Voyager` permite o multicast com o mecanismo de espaços e sub-espaços
- `Subspace`: container de distribuição dentro de uma mesma VM
- `Space`: container de distribuição que pode abranger múltiplas VMs - conexão de vários `Subspaces`



Sistema de Mensagens

■ Mecanismo de Publish-Subscribe

- objetos de um Space ou Subspace podem inscrever-se para receber eventos acontecendo no Space
- para tanto, eles devem implementar a interface `PublishedEventListener`, que define um único método `publishedEvent(event, topic)`. Este, é chamado quando alguém publica algum evento no Space ou Subspace
- um objeto pode ainda utilizar uma instância da classe `Subscriber` e inserí-la no Subspace
- para publicar um evento, utiliza-se o seguinte método estático da classe `Publish`
 - l `static void invoke(ISubspace subspace, java.util.EventObject event, Topic topic)`



Mobilidade e Agentes Móveis

■ Mobilidade

- capacidade de mover objetos de um servidor para outro

■ Agentes Móveis

- programas capazes de gerenciarem sua própria mobilidade

■ Classe Mobility

- provê métodos para mover qualquer objeto serializável
- para mover um objeto para uma nova localidade, utilize o método estático `Mobility.of()` para obter o facet de mobilidade do objeto e então usar os métodos definidos em `IMobility`

■ Interface Imobility

- `void moveTo(java.lang.Object destination)`
- `void moveTo(java.lang.String URL_destination)`



Mobilidade e Agentes Móveis

■ Efeito do moveTo

- qualquer mensagem que o objeto esteja correntemente processando ele continua a processar, porém não aceita novas mensagens
- o objeto e todas as suas partes não-transientes são copiadas para a nova locação usando a serialização do Java
- o novo endereço do objeto e suas partes não-transientes são cacheadas na locação antiga
- o objeto antigo é destruído
- mensagens suspensas para o objeto antigo são retomadas
- quando uma mensagem enviada via um proxy chega ao endereço antigo, ela é redirecionada para a nova locação
- o moveTo retorna após o objeto ter sido movido com sucesso ou, caso tenha havido alguma excessão, o objeto tenha sido restaurado a sua condição original



Mobilidade e Agentes Móveis

■ Notificação de Mudança

- Em alguns casos, pode ser necessário que o objeto faça arranjos antes e/ou depois da mudança
- para que isso seja possível, ele deve implementar a interface IMobile

■ Interface IMobile

- `void preDeparture(java.lang.String source, java.lang.String destination)`
- `void preArrival()`
- `void postArrival()`
- `void postDeparture()`



Agentes Móveis

■ Classe Agent

- provê métodos para mover um objeto que é um agente móvel
- para mover este objeto para uma nova localidade, utilize o método estático `Agent.of()` para obter o facet de agente do objeto e então usar os métodos definidos em `IAgent`

■ Interface IAgent

- `java.lang.String getHome()`
- `IResourceLoader getResourceLoader()`
- `boolean isAutonomous()`
- `void moveTo(java.lang.Object destination, java.lang.String callback)`
- `void moveTo(java.lang.Object destination, java.lang.String callback, java.lang.Object[] args)`
- `void moveTo(java.lang.String destination, java.lang.String callback)`
- `void moveTo(java.lang.String destination, java.lang.String callback, java.lang.Object[] args)`
- `void setAutonomous(boolean flag)`
- `void setResourceLoader(IResourceLoader resourceLoader)`



Mobilidade de Código

■ Disponibilizando os arquivos .class

- Pré-instalação de todos os arquivos .class no CLASSPATH do host remoto
- Manter todas as classes em um mesmo repositório, disponibilizado via uma URL
- Fazer com que o agente registre um "resource loader" antes de chegar. Desta maneira, o agente pode carregar consigo suas próprias classes, quando navega pela rede

■ Voyager: 2 implementações de IResourceLoader

- URLResourceLoader
- ArchiveResourceLoader



EC4 - Exercício Computacional 4

- Implementar a seguinte configuração
 - Implementar um objeto Market, que vende informações da seguinte forma:
 - | Lista arquivos, com preço e conteúdo em Kbytes
 - | Recebe agentes móveis para negociação
 - | Vende arquivos, retirando dinheiro do agente e retornando o arquivo desejado
 - Implementar um agente Trader, que compra informações da seguinte forma:
 - | Agente obtém do usuário a lista de arquivos que deseja comprar, uma lista de Markets que deseja visitar e uma quantidade de dinheiro
 - | Agente move-se para os Markets indicados, procurando pelos arquivos desejados
 - | Encontrando o arquivo, o agente compra-o, incorporando-o em seu código e retorna ao final da busca para sua origem



EC4- Exercício Computacional 4

- Fazer o seguinte teste
 - Criar 3 objetos Market em máquinas diferentes, com arquivos diferentes sendo vendidos, sendo que pelo menos 1 deles exista em mais de um Market
 - Criar um agente Trader e dar a ele a incumbência de encontrar 5 arquivos diferentes, sendo que 3 deles devem existir unicamente em cada servidor, um deles deve existir em 2 servidores e um deles não deve estar em nenhum servidor
 - Indicar ao agente Trader os Markets a visitar, e dar dinheiro suficiente para a compra de todos os arquivos
 - Verificar se o agente retorna com todos os arquivos que poderia comprar