



Open-source Education →

Atualização para Java 5

Iniciativa Globalcode

Agenda

1. Autoboxing;
2. Varargs;
3. Formatters / printf;
4. Scanners;
5. Static import;
6. Type-safe enumerations;
7. Enhanced for loop;
8. Generics;
9. Instrumentação;
10. Mais novos recursos;

Autoboxing

- Recurso para facilitar uso de wrapper classes;
- Exemplo “Moda antiga”:

```
public void oldway() {  
    int i = 10;  
    Integer objeto = new Integer(i);  
    if (objeto.intValue() == 10) {  
        System.out.println("Igual");  
    }  
}
```

Autoboxing

- Autoboxing para construção e comparação:

```
int i = 10;  
Integer objeto = i;  
if(objeto==10) {  
    //  
}
```

- Casting e pseudo mutabilidade

```
Integer objeto2 = (Integer) 1;  
objeto2 = 15;
```

Autoboxing

- Objetos como primitivos... Primitivos como objetos...

```
Boolean ehBissexto = true;  
if(ehBissexto) {  
    //  
}
```

Agenda

1. Autoboxing;
2. Varargs;
3. Formatters / printf;
4. Scanners;
5. Static import;
6. Type-safe enumerations;
7. Enhanced for loop;
8. Generics;
9. Instrumentação;
10. Mais novos recursos;

Var-args

- Excelente recurso para facilitar transmissão de múltiplos argumentos para métodos;

- Vamos imaginar o seguinte método tradicional Java:

```
public static void listaObjetos (Object objetos[]) {  
    for(int x=0;x<objetos.length;x++) {  
        System.out.println(objetos[x].getClass().getName()  
            + ", valor=" + objetos[x].toString());  
    }  
}
```

- Com este método, podemos variar o número de parâmetros através do array de objetos;

Var-args

- Exemplo de uso do método:

```
public static void main(String args[]) {  
    Object parametros[] = new Object[3];  
    parametros[0] = "Teste";  
    parametros[1] = new Integer(10);  
    parametros[2] = new java.util.Date();  
    listaObjetos(parametros);  
}
```

- Neste caso, estamos passando um array de 3 objetos.
Poderia ser 10, 1, 20, 1.000...

Var-args

- Var-args é portanto um recurso que facilita a escrita de código e principalmente uso de código para estas ocasiões;
- Vejamos um exemplo de método com var-args:

```
public static void listaObjetosVA(Object ... objetos) {  
    for(int x=0;x<objetos.length;x++) {  
        System.out.println(objetos[x].getClass().getName()  
            + ", valor=" + objetos[x].toString());  
    }  
}
```

- Na prática teremos um array de objetos exatamente igual ao exemplo anterior;

Var-args

Onde está a vantagem?

- A vantagem é para quem usa o método, veja a diferença:

```
Object parametros[] = new Object[3];  
parametros[0] = "Teste";  
parametros[1] = new Integer(10);  
parametros[2] = new java.util.Date();  
listaObjetos(parametros);
```

```
listaObjetosVA("teste", new Integer(10), new java.util.Date());  
listaObjetosVA("teste");  
listaObjetosVA(10);
```

Var-args

Conclusões:

- Em métodos declarados como (TipoObjeto ... objetos), podemos passar n argumentos, separados por vírgula;
- O próprio Java se encarregará de transformar os argumentos em um array de objetos;
- Podemos trabalhar com primitivos, Object ou então um tipo mais específico;
- Podemos sobrecarregá-lo;

Var-args

- Podemos trabalhar com primitivos, Object ou então um tipo mais específico; podemos sobrecarregá-lo;

```
public static void listaObjetosVA(int ... objetos) {  
    for(int x=0;x<objetos.length;x++) {  
        System.out.println(objetos[x]);  
    }  
}
```

```
public static void listaObjetosVA(Object ... objetos) {  
    for(int x=0;x<objetos.length;x++) {  
        System.out.println(objetos[x]);  
    }  
}
```

Agenda

1. Autoboxing;
2. Varargs;
3. **Formatters / printf;**
4. Scanners;
5. Static import;
6. Type-safe enumerations;
7. Enhanced for loop;
8. Generics;
9. Instrumentação;
10. Mais novos recursos;

Formatters

- java.util incorpora a classe Formatter que prove formatação estilo *printf*;
- Podemos formatar de formatar números, datas e Strings;

```
public static void main(String args[]) {  
    StringBuilder result = new StringBuilder();  
    Formatter formatter = new Formatter(result);  
    formatter.format("Exemplo de data: %1$tH:%1$tM",  
        new java.util.Date());  
    System.out.println(result.toString());  
    System.out.printf("printf do System.out %1$tH:%1$tM",  
        new java.util.Date());  
}
```

Número do argumento

Formato

Formatters

```
public static void main(String args[]) {  
    StringBuilder result = new StringBuilder();  
    Formatter formatter = new Formatter(result);  
    formatter.format("Exemplo de data: %1$tH:%1$tM",  
        new java.util.Date());  
    System.out.println(result.toString());  
    System.out.printf("printf do System.out %1$tH:%1$tM",  
        new java.util.Date());  
}
```

Exemplo de data: 19:22

Exemplo via printf do System.out 19:22

Agenda

1. Autoboxing;
2. Varargs;
3. Formatters / printf;
4. **Scanners;**
5. Static import;
6. Type-safe enumerations;
7. Enhanced for loop;
8. Generics;
9. Instrumentação;
10. Mais novos recursos;

Scanner

- Finalmente uma classe para representar um teclado!
- Simples e fácil:

```
public class Teclado {  
    public static void main(String args[]) {  
        Scanner teclado = new Scanner(System.in);  
        System.out.println("Digite o ano:");  
        int ano = teclado.nextInt();  
        System.out.printf("O ano digitado foi %s", ano);  
    }  
}
```



nextQualquerTipo()

Agenda

1. Autoboxing;
2. Varargs;
3. Formatters / printf;
4. Scanners;
- 5. Static import;**
6. Type-safe enumerations;
7. Enhanced for loop;
8. Generics;
9. Instrumentação;
10. Mais novos recursos;

Static import

- Um recurso para facilitar o uso de métodos e atributos estáticos de uma classe;
- Vejamos o seguinte exemplo de uso de rotina de Logging, com e sem static import, supondo o uso desta classe:

```
public class Logger {  
    public static final int DEBUG=0;  
    public static final int ERROR=1;  
  
    public static void log(String msg, int level) {  
        System.out.println(msg);  
    }  
}
```

Static import

```
import meulogging.Logger;
```

- Sem static import:

```
public class SemStaticImport {  
    public static void main(String args[]) {  
        Logger.log("Fácil, fácil...", Logger.DEBUG);  
    }  
}
```

```
import static meulogging.Logger.*;
```

- Com static import:

```
public class ComStaticImport {  
    public static void main(String args[]) {  
        log("Fácil, fácil...", DEBUG);  
    }  
}
```

Agenda

1. Autoboxing;
2. Varargs;
3. Formatters / printf;
4. Scanners;
5. Static import;
- 6. Type-safe enumerations;**
7. Enhanced for loop;
8. Generics;
9. Instrumentação;
10. Mais novos recursos;

Enumeration

- É um tipo de dado;
- Nova forma para criarmos tipos de dados finitos;
- Exemplos: dias da semana, comandos da tela, estado do registro do DB, categorias de um produto, cores, etc.
- Podemos criar dentro ou fora da classe;
- Podem ser simples ou complexas, com atributos e comportamentos;

Enumeration

- Exemplo de enumeration dentro de uma classe:

```
public class EnumInterna {  
    enum Estado { adicao, edicao, visualizacao };  
    Estado estado = Estado.adicao;  
}
```

- Ao compilarmos a classe, teremos dois arquivos:

```
1.EnumInterna.class  
2.EnumInterna$Estado.class
```

- Podemos dizer que uma enumeration é uma classe com dados pré-definidos;

Enumeration

- Exemplo de enumeration independente, fora da classe:

```
public enum Dia {  
    segunda, terça, quarta,  
    quinta, sexta, sabado,  
    domingo;  
}
```

- Ficará armazenada no arquivo `Dia.java` e após a compilação teremos `Dia.class`;
- É como uma classe, e também ficará em um pacote;

Enumeration

- Usando a enumeration Dia:

```
public class UsaDia {  
    public static void main(String args[]) {  
        Dia hoje = null;  
        hoje = Dia.quarta;  
        System.out.println(hoje);  
        if(hoje==Dia.quarta) {  
            System.out.println("Dia de fejuca!");  
        }  
    }  
}
```

Enumeration

```
public enum Comando {  
    save("globalcode.comando.Save"),  
    delete("globalcode.comando.Save"),  
    add("globalcode.comando.Save");  
    String classe;  
    Comando(String classe) {  
        this.classe=classe;  
    }  
    public void execute() throws Exception {  
        Object obj = Class.forName(classe).newInstance();  
        System.out.println("Executar objeto " + obj);  
    }  
}
```

Agenda

1. Autoboxing;
2. Varargs;
3. Formatters / printf;
4. Scanners;
5. Static import;
6. Type-safe enumerations;
- 7. Enhanced for loop;**
8. Generics;
9. Instrumentação;
10. Mais novos recursos;

Enhanced for

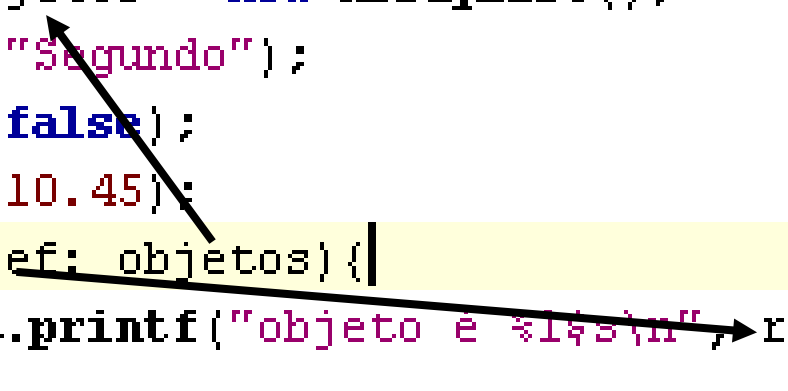
Forma tradicional de percorrer uma coleção:

```
public static void main(String args[]) {  
    ArrayList objetos = new ArrayList();  
    objetos.add("Segundo");  
    objetos.add(false);  
    objetos.add(10.45);  
    Iterator i = objetos.iterator();  
    Object ref = null;  
    while(i.hasNext()) {  
        ref = i.next();  
        System.out.printf("objeto é %1$s\n", ref);  
    }  
}
```

Enhanced for

Com Java 5...

```
public static void main(String args[]) {  
    ArrayList objetos = new ArrayList();  
    objetos.add("Segundo");  
    objetos.add(false);  
    objetos.add(10.45);  
    for(Object ref: objetos){  
        System.out.printf("objeto e %1$s\n", ref);  
    }  
}
```



Agenda

1. Autoboxing;
2. Varargs;
3. Formatters / printf;
4. Scanners;
5. Static import;
6. Type-safe enumerations;
7. Enhanced for loop;
- 8. Generics;**
9. Instrumentação;
10. Mais novos recursos;

Generics

- Mudança de maior impacto na linguagem;
- De uma visão “user-level”, facilita a manipulação de coleções, permitindo a restrição do tipo dos objetos da coleção;
- Para criadores de API's, Generics torna a P.O.O. do Java mais poderosa ainda;
- Requer todos seus conhecimentos sobre P.O.O., interfaces, polimorfismo, etc.

Generics

- Collection sem Generics...

```
public class UsoDeCollectionTradicional {  
    public static void main(String args[]) {  
        ArrayList clientes = new ArrayList();  
        Cliente cl = new Cliente();  
        clientes.add(cl);  
        //...  
        Cliente recuperado = (Cliente) clientes.get(0);  
    }  
}
```

O método add espera um
Object: add(Object o)

Generics

- Collection com Generics...

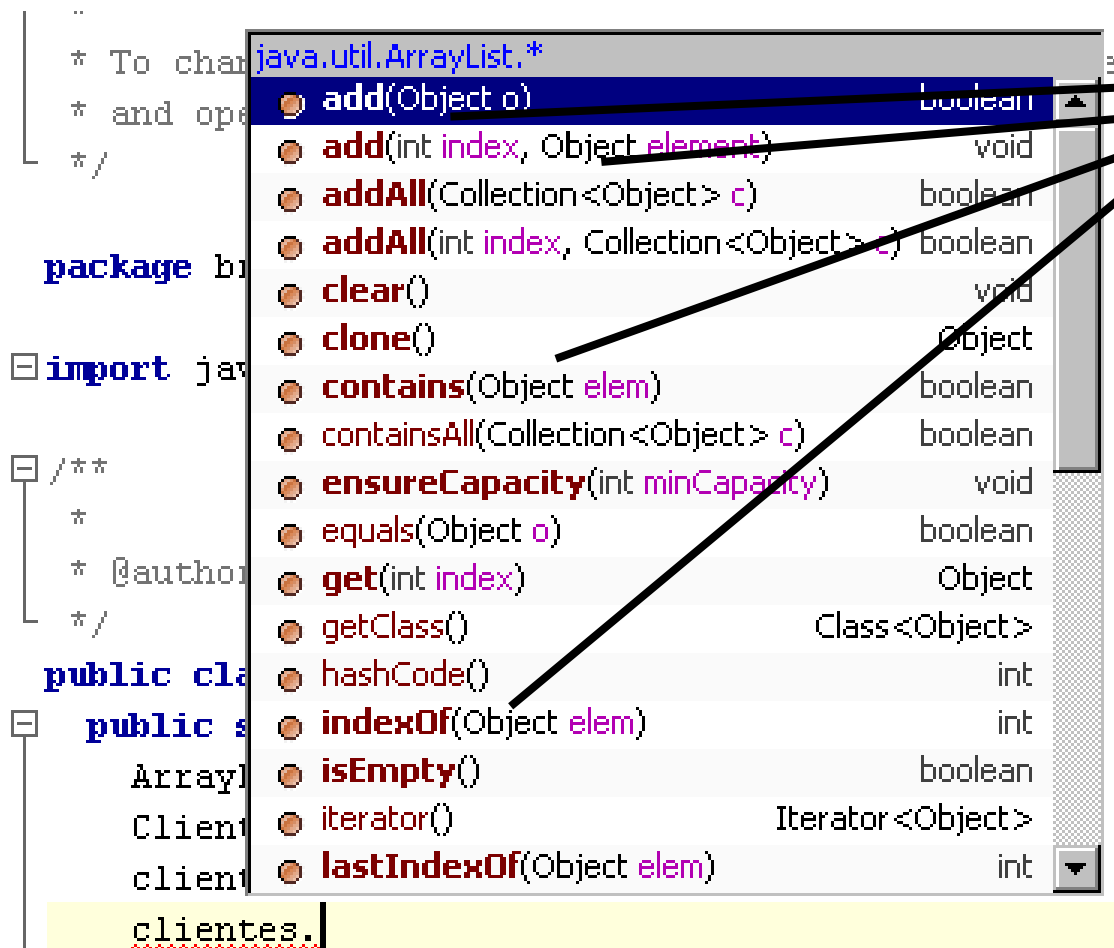
```
public class UsoDeCollectionGenerics {  
    public static void main(String args[]) {  
        ArrayList<Cliente> clientes = new ArrayList();  
        Cliente cl = new Cliente();  
        clientes.add(cl);  
        Cliente recuperado = clientes.get(0);  
    }  
}
```

Generics

O casting não foi necessário

- Informamos para a Collection que queremos trabalhar com Cliente, ou filhos de Cliente;

Generics



Sem Generics, os métodos da Collection ArrayList trabalham com Object

Generics

Com Generics, os parâmetros de ArrayList se adaptaram para trabalhar com Cliente

java.util.ArrayList<Cliente>.*		
add(Cliente o)		boolean
add(int index, Cliente element)		void
addAll(Collection<Cliente> c)		boolean
addAll(int index, Collection<Cliente> c)		boolean
clear()		void
clone()		Object
contains(Object elem)		boolean
containsAll(Collection<Object> c)		boolean
ensureCapacity(int minCapacity)		void
equals(Object o)		boolean
get(int index)		Cliente
getClass()		Class<Object>
hashCode()		int
indexOf(Object elem)		int
isEmpty()		boolean
iterator()		Iterator<Cliente>
lastIndexOf(Object elem)		int

```
clientes.
```

Generics

- Generics = é o parâmetro do parâmetro;
- O desenvolvedor de API's cria uma classe genérica;
- O usuário da API, usa a classe para um fim específico;
- Um estudo de caso...

Generics: estudo de caso

- Vamos imaginar a seguinte modelagem para DAO's:

```
public interface DAO {  
    public void save(Object o);  
    public void delete(Object o);  
    public Object getByID(int id);  
    public Collection getAll();  
}
```

Para mantermos o padrão de DAO, segundo a interface, trabalhamos com object e collections...

```
public class ClienteDAO implements DAO{  
    public void save(Object o) { }  
    public void delete(Object o) { }  
    public Object getByID(int id) {  
        return new ClienteBean();  
    }  
    public java.util.Collection getAll() {  
        return new java.util.ArrayList();  
    }  
}
```

Generics: estudo de caso

- Problemas deste código:

```
public class ClienteDAO implements DAO {  
    public void save(Object o) { }  
    public void delete(Object o) { }  
    public Object getByID(int id) {  
        return new ClienteBean();  
    }  
    public java.util.Collection getAll() {  
        return new java.util.ArrayList();  
    }  
}
```

1. save e delete podem receber qualquer coisa;
2. O getByID e getAll podem retornar qualquer coisa;
3. Tudo para manter os DAO's do aplicativo padronizados;

Generics: estudo de caso

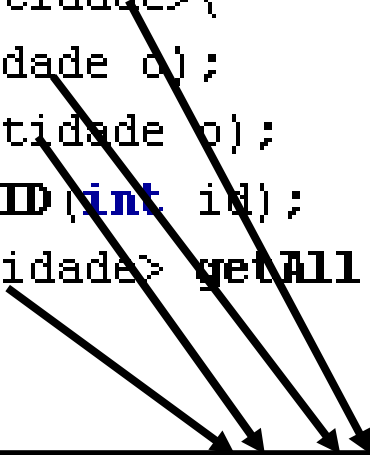
- Para usar este DAO:

```
public static void main(String args[]) {  
    ClienteDAO dao = new ClienteDAO();  
    ClienteBean c = (ClienteBean) dao.getByID(10);  
    //o save recebe cliente como Object, polimorfimo.  
    dao.save(c);  
    //por este motivo podemos também passar uma String...  
    dao.save("TESTE");  
}
```

Generics: estudo de caso

- Agora com Generics...

```
public interface DAO<Entidade>{  
    public void save(Entidade o);  
    public void delete(Entidade o);  
    public Entidade getByID(int id);  
    public Collection<Entidade> getAll();  
}
```



Especificamos que a entidade será genérica.

Neste caso, qualquer filho de Object.

Generics: estudo de caso

- Implementamos a interface...

```
public class ClienteDAO implements DAO {  
    public void save(Object o) {  
    }  
    public void delete(Object o) {  
    }  
    public ClienteBean getByID(int id) {  
        return null;  
    }  
    public java.util.Collection<ClienteBean> getAll() {  
        return null;  
    }  
}
```

Temos que receber
como Object...

A covariância permite implementarmos um
tipo de retorno mais específico que Object.

Generics: estudo de caso

- Uso mais natural e simples...

```
public static void main(String args[]) {  
    DAO<ClienteBean> dao = new ClienteDAO();  
    ClienteBean c = dao.getByID(10);  
    dao.save("String nao funciona");  
}
```

- Vamos usar a interface DAO com objetos ClienteBean e implementação ClienteDAO;
- Não precisamos fazer casting e não podemos enviar Strings para o save e delete;

Agenda

1. Autoboxing;
2. Varargs;
3. Formatters / printf;
4. Scanners;
5. Static import;
6. Type-safe enumerations;
7. Enhanced for loop;
8. Generics;
- 9. Instrumentação;**
10. Mais novos recursos;

Instrumentação

- JMX – Java Management Extension é um padrão para gerenciar recursos como aplicativos, devices e serviços;
- Diversas classes de controle e serviços da máquina virtual 1.5 adotaram o padrão JMX;
- Um serviço gerenciado é chamado de Managed Bean ou MBean;
- Um servidor é um conjunto de MBean's, chamado de MBean Server;

Instrumentação

- Podemos monitorar um MBean server através de conectores;
- Para monitorar um o MBean server da VM, devemos configurar o seguinte parâmetro:

```
java -Dcom.sun.management.jmxremote  
-jar SwingSet2.jar
```

- Neste caso ligaremos o aplicativo SwingSet2.jar com recursos de monitoração de VM;

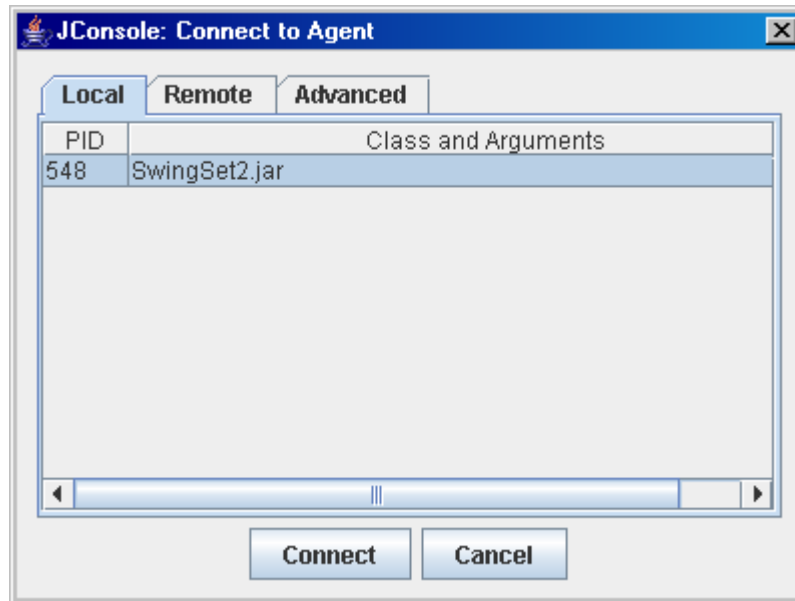
Instrumentação

- Agora ligaremos a ferramenta de monitoração embutida no JDK 5, digite jconsole:

```
c:\>jconsole    ou    [root@java root]jconsole
```

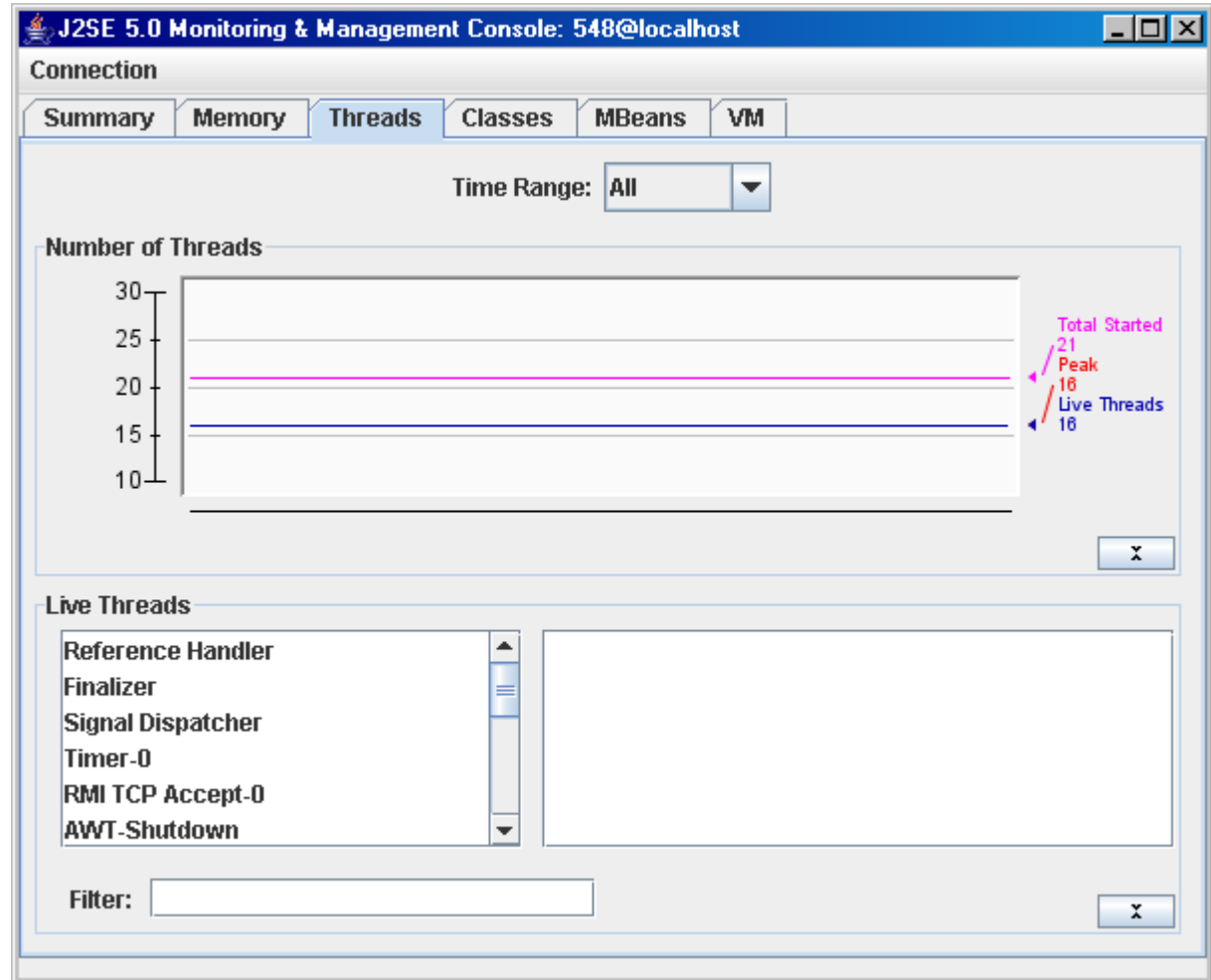
Instrumentação

- A seguinte janela deve aparecer, apresentando uma lista das VM's que podem ser monitoradas:



Instrumentação

- Memória, Thread, classes e seus próprios MBeans podem ser monitorados



Instrumentação

- Você pode criar objetos para embutir neste sistema de monitoração com baixo custo de desenvolvimento;
- Basicamente terá que desenvolver uma interface MBean e efetuar o registro do objeto no MBean Server;
- Maiores informações:

JMX Tutorial

<http://java.sun.com/j2se/1.5.0/docs/guide/jmx/tutorial/tutorialTOC.html>

Agenda

1. Autoboxing;
2. Varargs;
3. Formatters / printf;
4. Scanners;
5. Static import;
6. Type-safe enumerations;
7. Enhanced for loop;
8. Generics;
9. Instrumentação;
10. Mais novos recursos;

Outros recursos...

- Annotations / Metadata: facilita a escrita de código repetitivo e excessivo (RMI, JDBC, EJB, Servlets, Hibernate, MBeans);
- Annotation pode gerar código;
- XDoclet = Annotation;
- Novos recursos no BigDecimal;
- API de programação concorrente / thread foi aprimorada com a API do Doug Lea;

Outros recursos...

- Novos recursos no BigDecimal;
- API de programação concorrente / thread foi aprimorada com a API do Doug Lea;
- Class Data Sharing: classes de sistema são compartilhadas por múltiplas máquinas virtuais;
- GC: podemos especificar objetivos de performance para a VM ajustar a memória Heap automaticamente;

Outros recursos...

- Todos os exemplos estão disponibilizados na URL:

www.globalcode.com.br/private/mc19.zip

- URL Java 5: java.sun.com/j2se/1.5.0
- Agradecimentos ao Glaucio do GUJava/SC pela colaboração com material;



Open-source Education →

Atualização para Java 5

Iniciativa Globalcode