UBS

J2EE Consulting

Java Programming Guidelines and Best Practices

Version 1.1, April 2006

For Bank Internal Use Only

Collection of guidelines and recommendations for Java programming

Author	....	P. Collovà
Reviewer(s):	....	J2EE Consulting Team
Approval:	....	Th. Leemann
Distribution:	Java developers at UBS-IT	

Table of Contents

1. Introduction 1
1.1. Purpose and Scope 1
1.2. Prerequisites 1
1.3. References 2
2. Coding Guidelines 5
2.1. Code Layout 5
2.2. Code Documentation 7
2.2.1. How to Use Javadoc™ 8
2.3. Code Structure 10
2.3.1. Derive Packages from Design 10
2.3.2. General Class Structure 11
2.3.3. Applying Design Patterns 17
2.3.4. Programming by Contract 19
2.3.5. Rules for Constructors, Object Creation and Destruction 24
2.3.6. Rules for Methods 26
2.3.7. Methods from java.lang.object and JDK 28
2.3.8. Hints for Expressions and Control-Flow Structures 34
2.3.9. Best Practices in Exception Handling 37
2.4. Performance and Optimization 40
3. Use of Specific Java Mechanisms 44
3.1.1. Concurrency and Synchronization 44
3.1.2. Serialization 45
4. Solutions to Specific Problems, Use of Libraries 46
4.1.1. Data Structure Implementation, Collections 46
4.1.2. Date and Time 53
4.1.3. String Handling 58
4.1.4. Regular Expressions 60
4.1.5. Mathematics 64
Appendix 1 69
Appendix 2 70

Revision History

Version and date	Change description
Version 1.0, Draft A - May 19, 2006	First issued version
Version 1.1, Final - April 12, 2006	Approved by J2EE Consulting, UBS TA and Java CPM

1. Introduction

1.1. Purpose and Scope

The following guidelines and best practices are based on knowledge from practice, information exchange, books, articles and papers. The whole was condensed and partially amended using collective experience made during development and consulting.

The goal is not to replicate existing knowledge, but rather to condense and concentrate it to a set of rules and guidelines to be consulted during software development. To avoid reading hundreds of pages or asking many times experts for help, useful knowledge is presented in form of a quick reference, a kind of "good Java programming in a nutshell".

In-depth explanations are not part of the guidelines. The reader is urged to consult the literature for the reasons behind them. This document tries to organize the existing knowledge and to abstract it into a set of rules for guidance. References are provided for further reading, in order to satisfy those readers interested in the rationale behind specific rules and hints.

This document should be considered as a suggestion to reason about the own code (and the underlying design), not as a set of laws to be followed blindly and slavishly.

The document will be helpful, if it helps programmers spend their time on solving the problems of their customers, instead of worrying about technical details.

1.2. Prerequisites

The reader should at least have a previous exposure to programming in general, and in particular to Java. A basic understanding of object-oriented technology, as well as of data structures and algorithms is required for a good understan-ding of the guidelines.

1.3. References

The publications listed below are recommended for further reading. They served as basis for the guidelines. The reference [3] is highly recommended as first lecture.

Ref	Document	Version and Reference No
[1]	VERMEULEN/et alii, The Elements of Java Style	Cambridge University Press, 2000 ISBN 0-521-77768-2
[2]	UBS Application Development Guide + SSP WebSphere Naming Conventions	http://bw.docweb.it.ubs.ch/doc/livelink/Application_Development_Guide_2.4?func=doc.Fetch&nodeId=8146010&docTitle=Application+Development+Guide+2%2E4&viewType=1
[3]	BLOCH, Effective Java	Addison-Wesley, 2001 ISBN 0-201-31005-8 http://java.sun.com/docs/books/effective
[4]	HAGGAR, Practical Java. Programming Language Guide	Addison-Wesley, 2000, *ISBN 0-201-61646-7*
[5]	WILSON/KESSELMAN, Java Platform Performance. Strategies and Tactics	Addison-Wesley, 2000, *ISBN 0-201-70969-4*
[6]	LARMAN/GUTHRIE, Java 2 Performance and Idiom Guide	Prentice-Hall, 2000, *ISBN 0-130-14260-3*
[7]	UBS J2EE Presentation Layer including: Struts@UBS, UBSWidgets	J2EE Presentation Layer Homepage

The following references cover the topic of concurrent programming, a complex domain requiring in-depth knowledge and experience in operating systems, process and memory management.

Ref	Document	Version and Reference No
[C-1]	LEA, Concurrent Programming in Java. Design Principles and Patterns	Addison_Wesley, 1997 *ISBN 0-201-69581-2*
[C-2]	OAKS/WONG, Java Threads	O'Reilly, 1999, 2nd edition *ISBN 1-565-92418-5*

Ref	Document	Version and Reference No
[C-3]	HOLUB, Taming Java Threads	Apress, 2000 *ISBN 1-893-11510-0*
[C-4]	GARG, Concurrent and Distributed Computing in Java	Wiley, 2004 *ISBN 0-471-43230-X*

Among a lot of other ones, the following links contain references to interesting articles of good quality on Java topics (some are general, others are restricted to specific topics). It is recommended to visit these sites from time to time.

Ref	URL
[URL-1]	www.javaworld.com General articles on Java technologies
[URL-2]	www.onjava.com General articles on Java technologies
[URL-3]	www.precisejava.com Tutorials
[URL-4]	java.almanac.com Examples about the JDK classes
[URL-5]	www.crossroads.com/bradapp/javapats.html community.java.net/patterns Overview of design patterns
[URL-6]	java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html Introduction to assertions
[URL-7]	http://www.regular-expressions.info/index.html (Web site devoted to regular expressions) www.regexlib.com (examples) www.regex.info (an excellent book on REs) ➔ the book presented on this site is FRIEDL, Mastering Regular Expressions, O'Reilly, 2nd ed.
[URL-7]	www2.hursley.ibm.com/decimal in-depth information about decimal numbers and floating-point computations http://java.sun.com/developer/JDCTechTips/2003/tt0204.html#2 paper about floating-point arithmetic in Java

A lot of books cover the topic of "Software Design Patterns". Among others, the following are of interest for Java programmers (the first one is the famous "GoF book"):

Ref	Document	Version and Reference No

Ref	Document	Version and Reference No
[D-1]	GAMMA/et alii, Design Patterns. Elements of Reusable Object-Oriented Software	Addison-Wesley, 1997 *ISBN 0-201-63361-2*
[D-2]	STELTING/MAASSEN, Applied Java Patterns	Prentice Hall, 2001 *ISBN 0-130-93538-7*
[D-3]	METSKER/WAKE, The Design Patterns in Java Workbook	Addison-Wesley, 2002 *ISBN 0-201-74397-3*
[D-4]	ALUR/et alii, Core J2EE Patterns. Best Practices and Design	Prentice Hall, 2003 *ISBN 0-131-42246-4*

Concerning algorithms and data structures, and given the fact that implementations are available through the Java Collections suite, any book is possibly suitable for self-teaching, even if not using directly Java.
However, if one wants to see the application of data structures and algorithms in Java, using the most advanced features of Java 5, the following book is suitable:

Ref	Document	Version and Reference No
[A-1]	KOFFMANN/WOLFGANG, Objects, Abstraction, Data Structure and Design Using Java Version 5.0	Wiley, 2004 *ISBN 0-471-69264-6*

2. Coding Guidelines

2.1. Code Layout

The purpose of guidelines on formatting is to produce code which can be read and understood with ease by every developer in and outside the company.

The goal is to produce source code which is readable by many persons, not to define strict numerical values for a single layout attribute (e.g. "indent by 4 spaces"). A consistent layout is an important help in team communication and a small, but effective step toward "ego-less programming".

Many of these rules could be automatically enforced in the development environment, without manual intervention. Eclipse/RAD offers a large array of options in this regard.

The guidelines in this chapter are based on [1].

Code layout

R.LAY.1	Indent nested code.
R.LAY.2	Break up long lines.
R.LAY.3	Include white space.
R.LAY.4	Do not use "hard" tabs, let the editor (or the formatter/beautifier) replace them by spaces.
R.LAY.5	Write only one statement or one declaration per line.

Naming conventions

R.NAM.1	Apply the UBS conventions first (see [2]). If a case is not covered by these conventions, apply the following guidelines. In case of collisions, the UBS conventions must be used.
R.NAM.2	Use meaningful, familiar names, not single letters. Avoid too long names, but use complete ones.
R.NAM.3	Avoid abbreviations and acronyms, excepted for well-known ones.
R.NAM.4	Capitalize only the first letter in acronyms.
R.NAM.5	Do not use names that differ only in case or grammatical number (singular, plural).
R.NAM.6	Apply the rules in [2] for package names.
R.NAM.7	Use the widely known "camel case" notation:

- classes, interfaces: first letter of every word capitalized, rest of name in lowercase.

- attributes, methods: first letter lowercase, first of every following word capitalized, rest of name in lowercase.
- constants: all uppercase, with words separated by underscore.

R.NAM.8 Use nouns when naming classes and attributes.

R.NAM.9 Use verbs when naming methods.

R.NAM.10 Use standard names for trivial throwaway variables (but only for these), to be used in the smallest possible scope (e.g. `for` statement). Possible names are as follow:

• character	`c`
• int	`i`
• String	`s`
• Object	`o`

At least `i` in loops is used very frequently.

R.NAM.11 Apply the naming conventions for variables to arrays, too. Place the brackets directly adjacent to the type itself.

R.NAM.12 Follow JavaBean naming conventions for names of accessor and mutator methods (`get/set; is/has` for Boolean variables).

R.NAM.13 Use the following prefixes and postfixes for typical classes:

- *Abstract* (prefix) for abstract classes.
- *Default* (prefix) for the default implementation of an abstract class.
- *<verb>* ¦ *<noun>* + '*able/ible*' for interfaces.
- *Factory* (postfix) for implementations of the Abstract Factory pattern
- '*s*' (postfix) for collections (e.g. `customers`)

R.NAM.14 When using the Struts framework, follow the guidelines of Struts@UBS [7]:

- Use the postfix *Action* for classes extending the Struts Action class: *<class_name>* + *Action*
- Use the postfix Form for classes extending the Struts *ActionForm* class: *<class_name>* + *Form*
- Use the postfix *Form* for Struts *FormBeans*: *<bean_name>* + *Form*

2.2. Code Documentation

The primary purpose for documentation comments is to define a programming contract between a client and a supplier of a service.

The purpose of guidelines on documentation is to enhance and extend the information contained in the design documentation (UML diagram) by additional information related to the whole code and single parts of it.

Doing so, a bridge between design and implementation can be drawn, which helps reconstructing the way taken and the technical decisions made during the transaction from design to code.

General guidelines

R.DOC.1	Use (American) English exclusively.
R.DOC.2	Write documentation for those who must use your code and those who must maintain it.
R.DOC.3	Keep comments and code in synchronization. When code is changed, the corresponding documentation must change, too.
R.DOC.4	Use the active voice and omit needless words.
R.DOC.5	Describe the programming interface before writing the code.
R.DOC.6	Use a single consistent format and organization for all documentation comments.
R.DOC.7	Write summary descriptions that stand alone. Omit the subject when describing actions, subject and verb when describing things.
R.DOC.8	Include examples in the documentation.
R.DOC.9	Document known defects and deficiencies.
R.DOC.10	Document preconditions, postconditions, and invariants.

R.DOC.11	Describe "why" the code is doing what it does, not "what" the code is doing. That means avoiding redundant comments (such es `i++; // increment by one`), concentrating on the explanation of complex algorithms and non-evident solutions.
	Code should never be documented to explain what the code is doing; the code itself already does that. Documentation should explain why a block of code is doing what it does. Anyone reading the code should be already familiar with the language used.

Comment structure

R.DOC.12	Use documentation comments to describe the programming interface. The documentation associated with a method should describe all aspects of behavior on which a caller of that method can rely and should not attempt to describe implementation details.
R.DOC.13	Provide a summary description and overview for each package.

R.DOC.14 Provide a summary description for each class, interface, field and method.

R.DOC.15 Fully describe the signature of each method.

R.DOC.16 Document all methods: `public, protected, package` (default visibility), and `private`.

R.DOC.17 Use a tag to emphasize package visibility (which is the default, if no modifier is used):

```
/* package-private */
```

R.DOC.18 Add internal comments only if they will aid others in understanding your code. Use one-line comments to explain implementation details.

Practical hints

R.DOC.19 Avoid the use of end-line comments, but explain local variable declarations with an end-line comment.

R.DOC.20 Establish and use a set of keywords to flag unresolved issues.
In Eclipse/RAD, use `//TODO` to mark open issues within the code.

R.DOC.21 Use standard comments to hide code without removing it (this feature is supported by Eclipse/RAD).

2.2.1. How to Use Javadoc™

The `javadoc` command-line utility is used to generate HTML pages from the documented code. Please consult the documentation to learn how to use it in the Eclipse/RAD environment.

R.JDC.1 Establish and use a fixed ordering for Javadoc™ tags.

R.JDC.2 Use the summary description and overview for each package (see chapter "Comment Structure") to generate the package documentation:

1. Create a package comment file named `package.html` (fixed name), and place that file in the package directory along with the source files
2. The file must contain HTML tags only, but not Java code
3. The description must appear within an HTML `<body>` element
4. The first sentence of the body text will be used as summary description for the package
5. All Javadoc tags, except for `@link`, can be used in this text. Any `@see` tag must use fully qualified names.

R.JDC.3 Provide a general (package-independent) overview:

1. Use it to document a group of related packages (in the sense of a subsystem), or for an application

2. Create an overview comment file. Although the name is arbitrary, these guidelines request the fixed name `overview.html` (as analogy to the package overview).
3. Place this file at the top of the source directory hierarchy, and use the `-overview` option of the `javadoc` tool to include it in the generated documentation

All criteria described above for `package.html` apply to this file, too.

R.JDC.4 Layout and format for Javadoc comments:

1. Start every comment line with `*`
2. Use a single space (no tabs) before the beginning of the text
3. Insert a blank line between the end of every section of descriptive text, between tags or between text and tags
4. Use the `<code>` tag to wrap single names or short code portions (no more than one line)
5. Use the `<pre>` tag to wrap code fragments
6. Use the `{@link #IDENTIFIER}` to insert references to documentation elements of interest to the reader
7. Don't use the `<tt>` tag, because it is a font tag and the use of such elements is discouraged in HTML 4.01 in favor of style sheets
8. Use character entities when representing characters outside the English alphabet: use `&lt;` for `'<'`, `&gt;` for `'<'`, `&amp;` for `'&'`.
   Be careful *not* to include a period within the first sentence of a documentation comment. It would prematurely terminate the summary description. As a consequence, avoid using abbrevia-tions, decimal fractions and similar. If really needed, code the period as `&#46;`
9. Use the templates below for Javadoc-based comments.

Header for classes and interfaces

```
/**
 * CLASS_NAME - DESCRIPTION
 *
 * project   PROJECT_NAME
 * created   DATE
 * modified  DATE
 * @author   UNIQUE_ID
 * @version  MAJOR.MINOR
 *
 * @see REFERENCE
 *
 * Copyright © YEAR by UBS AG.
 * All rights reserved.
 *
 */

Hint: the @see tag is optional.
```

Comment for methods

```
/**
 * DESCRIPTION
 *
 * @param  NAME PURPOSE
 * @return TYPE
 * @throws EXCEPTION
 * @see    REFERENCE
 *
 */

Hint: the @see tag is optional.
```

2.3. Code Structure

The purpose of guidelines on coding is to promote a consistent and simple style, which can help enhancing the maintainability and robustness of code. A set of rules about the basic aspects of coding can help avoiding subtle errors and let the developer concentrate on the implementation of the structures postulated during design, as well on the domain-specific algorithms.

2.3.1. Derive Packages from Design

R.CDE.1 Capture high-level design and architecture as stable abstractions organized into stable packages.

R.CDE.2 Maximize abstraction to maximize stability. The more abstract a package is, the more stable it tends to be.

R.CDE.3 Isolate volatile classes and interfaces in separate packages.

R.CDE.4 Avoid making packages that are difficult to change dependent on packages that are easy to change. A package should only depend on packages more stable than it is.

R.CDE.5 There must be no cycles in the dependency structure between packages (directed acyclic graph[1]).

R.CDE.6 Place types[2] that are commonly used, changed, and released together, or mutually dependent on each other, into the same package. A package consists of classes reused together.

R.CDE.7 Follow the Open-Closed Principle[3]: software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

[1] A directed graph is a graph with no path that starts and ends at the same vertex. Consult a book on discrete mathematics and graph theory for further information.

[2] A *type* is a collection of objects having similar structure. For instance, integers, pairs of integers and functions over integers are at least three distinct types. These divisons are called types, following the vocabulary of the mathematical type theory.

R.CDE.8 Follow the Liskov Substitution Principle[4]: methods that use references to base classes must be able to use objects of derived classes without knowing it. Consequence: define subclasses so they may be used anywhere their superclasses may be used.

2.3.2. General Class Structure

General advices

R.CDE.9 Follow the fundamental tenets of software design: encapsulation (information hiding), loose coupling, high cohesion.

R.CDE.10 Always use the same declaration order for a class body.
Two possible sequences are meaningful:

- order by increasing privacy (public parts before)
- order by "data first, operations last".

Choice a structure and use it in *every* class.

R.CDE.11 Always keep related parts together:

- constants, class variables, static initialization blocks
- instance variables, instance initialization blocks, constructors
- class methods, instance methods
- member types, helper classes.

R.CDE.12 Make each class or attribute as inaccessible as possible. Make all attributes private. Use protected attributes sparingly.

R.CDE.13 Favor composition[5] over inheritance. If using inheritance, follow these principles:

- design and document for inheritance or else prohibit it.
- be careful when a class invokes any of its overridable methods
- keep inheritance hierarchies small

[3] The Open-Closed Principle (defined by Bertrand Meyer in 1988) is one of the fundamental principles of object-oriented programming. Software entities (classes, modules, functions) should be stable (meaning "not changeable"), but at the same time flexible (meaning open to changes required by the customer). Flexibility means openess to extensions, without changes to the software entity itself (closeness).

[4] Liskov Principle (after Barbara Liskov): "If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T".

[5] This design is called *composition* because the existing class becomes a component of the new one. Each instance method in the new class invokes the corresponding method of the contained instance and returns the results. This is known as forwarding. Sometimes, this combination is called delegation. Hower, in a delegation, the wrapper object passes itself to the wrapped object (see GoF for further details).

- inherit behavior, not state.

R.CDE.14 Use polymorphism instead of `instanceof`.
Don't write such code:

```
if (liquid instanceof Milk)
{
    ((Milk) innerLiquid).swirlLikeMilk(false);
}
else if (liquid instanceof Coffee)
{
    ((Coffee) innerLiquid).swirlLikeCoffee(false);
}
else
{
    innerLiquid.swirlLikeGenericLiquid(false);
}
```

R.CDE.15 Wrap general-purpose classes that operate on `java.lang.Object` to provide static type checking.
☞ The new features (called "generics"[6]) in Java 5 will make this rule obsolete.

R.CDE.16 Define small classes and small methods. Replace repeated non-trivial expressions with equivalent methods. Factor out common functionality.

R.CDE.17 Apply refactoring, and put common code either in helper classes, if they are not intimately bound to the class using them.
If there is a strong relationship between class and helper class, which makes the use of the helper class only meaningful when close to the use of the class itself, use an inner class:

```
public class <CLASS>
{
     …
     class <INNER_CLASS>
     {
         …
     }
     …
}
```

An inner class only makes sense it if is going to be associated to the class that contains it. One good example for an inner class are iterators in collections.

If the inner class is not declared as static, it has full access to private attributes and methods of the enclosing class.

Interfaces

R.CDE.18 Prefer interfaces to abstract classes. Think of an interface as the expression of a contract to be implemented in a concrete class.
As a convenience, provide an abstract skeletal (partial) implementa-

[6] Look at http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html for a first introduction to the topic.

tion class for every interface (called `Abstract<NAME>`).
If an abstract class only contains method signatures, it should be declared as interface.

R.CDE.19 Interfaces are generally public, sometimes with package scope (no modifier). Don't use `static` or `abstract` modifiers, because they are redundant. Method declarations in interfaces are by definition `public` (but are restricted to package visibility if the interface itself has package visibility).

R.CDE.20 Use interfaces to support multiple inheritance, but take into account the problems related to multiple inheritance – use this mechanism sparingly, if at all.
Use one abstract class and one or more interfaces to implement *mixins*. A mixin is a type (typically an interface) that a class can implement in addition to its "primary type" to declare that it provides some optional behavior. This technique, also known as *simulated multiple inheritance*, provides most of the benefits of multiple inheritance, while avoiding the pitfalls.

R.CDE.21 Use a marker interface to signal that any class implementing this interface has the same property.

☞ The annotation mechanism in Java 5 will make this rule obsolete.

R.CDE.22 Avoid name clashes in interfaces (methods with the same name in different interfaces).

Immutability

R.CDE.23 Try to make classes immutable, whenever possible. Immutable objects are inherently thread-safe and can be shared freely.
If an immutable class implements `Serializable` and it contains attributes referring to mutable objects, provide an explicit `readObject()` or `readResolve()` method.

R.CDE.24 Consider declaring classes representing immutable data types as `final`.

R.CDE.25 Immutability can be achieved by:

a) not implementing accessor methods at all and making the class final

b) declaring an immutable interface that the mutable class implements

c) providing an interface containing only immutable methods, a non-final class that provides a mutable implementation, a final class that provides an immutable interface

d) using an immutable delegation class (of type `final`), containing only immutable methods.

How to make an already existing class immutable (e.g. if the source code is not available or the class cannot be changed):

```
public class <DELEGATE>
   implements <READ_INTERFACE>
```

```
{
   private <NON_IMMUTABLE> core;

   public <DELEGATE>( <NON_IMMUTABLE> core )
    {
       this.core = core;
    }

   public get<ATTRIBUTE>()
    {
       return core.get<ATTRIBUTE>();
    }
}
```

Import declarations

R.CDE.26 Keep import declarations ordered. Use the following order: first the Java packages (`java.<package>, javax.<package>`), then the packages from other products (`com.<package>`).
The application-specific packages follow at the end of the import list. Subpackages (`<package>.<subpackage>`) should directly follow the corresponding package declarations.

R.CDE.27 `java.lang` is automatically imported. There is no need to declare it explicitly.

R.CDE.28 Every class extends `java.lang.Object`. There is no need to mention explicitly the root class of all objects.

Constants

R.CDE.29 Declare constants as `private static final`. Apply the naming conventions, and write the constant name in uppercase, with underscores as separators.

If really needed, define public constants.

Don't define arrays of constants, because arrays are by definition mutable.
A better solution is to provide accessor methods for them:

```
private static final T[] PRIVATE_VALUES = { … };
public static final List VALUES =
   Collections.unmodifiableList(
               Arrays.asList( PRIVATE_VALUES ));
or

private static final List VALUES =
   Collections.unmodifiableList(
               Arrays.asList( PRIVATE_VALUES ));

public static List getConstants()
{
   return VALUES;
}
```

R.CDE.30 Don't use interfaces only to define and collect constants. Interfaces are for types only (see [3]).
Apply the following rules:

- If the constants are tied to a class or interface, add them to it (principle of localization).

- If you want to avoid defining a class for typed constants, use a non-instantiable utility class for "global" or "application-wide" constants. It acts as a single place for such constants.

```
public class <NAME>Constants
{
    private <NAME>Constants {}
    public static final <TYPE>
           <CONSTANT_NAME> = <VALUE>;
}
```

- If there are members of an enumerated type, use type-safe enumerations:

  ☞ The new features in Java 5 (`enum` as data type[7]) will make this rule obsolete.

The following code example shows the basic template for an enumeration class. Each single element of the enumeration (which corresponds mathematically to a set), is realized as pre-instantiated class.

Usage:

- declaration:
  `<METHOD>( <NAME> enum-element );`

- call sequence:
  `<CLASS>.<METHOD>( <NAME>.<CONSTANT_n> );`

```
public final class <NAME>
{
    // set of constants 1..n
    public static final <NAME>
        <CONSTANT_1> = new <NAME>(…);
    …
    public static final <NAME>
        <CONSTANT_n> = new <NAME>(…);

    // internal structure
    private transient <TYPE> <ATTRIBUTE>;
    private <NAME>(…) { … }

    // basic methods from java.lang.Object
    public final boolean equals( Object o )
    {
       return super.equals( o );
    }

    public final int hashCode()
```

[7] See http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Enum.html for the description of base class `Enum` and http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html for a short introduction.

```
    {
       return super.hashCode();
    }

    // support for serialization
    private static final <NAME>[]
        PRIVATE_VALUES = { … };
    public static final List VALUES =
        Collections.unmodifiableList
        (Arrays.asList( PRIVATE_VALUES ));

    private Object readResolve()
        throws ObjectStreamException
    {
        return PRIVATE_VALUES[order];
    }
}
```

Hint 1: If additional behavior is required, add specific methods to the enumeration class. Sometime such enumeration elements are called "smart constants", because they combine an immutable nature with additional functionality.
Hint 2: If hierarchies of enumerations are required by design, omit the keyword `final` from the class declaration.
Hint 3: How to introduce a "natural" order and support for comparison (the mechanism is not thread-safe!):

```
    // natural order capabilities
    private static int nextOrdinal = 0;
    private final int order = nextOrdinal++;

    public int compareTo( Object o )
    {
        return order - ((<NAME>)o).order;
    }
```

Hint 4: using a protected constructor, removing the `final` class modifier and the support for natural ordering allows for an extensible variant (a set of constants which can be extended by another set, providing a union of both sets).

R.CDE.31 Use a `static { … }` block for complex initializations at class-level only:

```
private static Map map = new HashMap();
static
{
        map.put( <KEY>, <VALUE> );
        …
}
```

R.CDE.32 Nonstatic code blocks are an extension of the initialization mechanism of instance variables. They are called at the time the instance variable's initializers are evaluated (that is, after construction of the superclass), in the order they appear in the source code.

```
{
        <ATTRIBUTE> = <VALUE>;
```

```
            ...
    }
```

Note the absence of any keyword: the non-static block is only demarcated by the braces.
Take care about possible class-loading issues or deadlocks.

2.3.3. Applying Design Patterns

The aim of this chapter is not to give an introduction to the wide topic of "Design Patterns", but to give guidelines about their meaningful use. The goal is to summarize major points which should be considered before applying patterns. Using design pattens is a good thing, but it can also misused, as any other programming technique.

For further information, please consult the literature reference at the beginning of the document (items [D-1] to [D-4]).

When applying design patterns, some major critical points should be considered:

- Not all patterns are simple to use and are free of side effects. Some should be used with care, because of technical consequences (consider the Singleton pattern in distributed environments) or implications (security, unwanted flexibility, complicated code).
- A pattern can be used the wrong way. The idea can be misunderstood, or it can be wrongly implemented.
- Overusing design patterns. Patterns should only be used when they improve the design.

Abstract Factory Pattern

R.PAT.1 As alternative to the constructor, a factory method (borrowing the idea from the *Abstract Factory pattern*) can be used. This public static method, usually named `getInstance()` or `valueOf()`, returns an instance of the class itself. However, it is not required to create an object on every invocation. Additionally, it can return an object of every subtype of their return type. A factory method, together with a private constructor, does not allow subclassing. A factory method can be used if computationally expensive validations must be carried out on the input parameters and one can assume that the checks can frequently fail. Object creation is then only done when all validations succeed. For this purpose, a simple and therefore efficient (not computational expensive) constructor can be used. Unnecessary memory allocation is avoided, because the object is only instantiated if the input is valid.

Adapter Pattern

R.PAT.2 *Adapter pattern*: used for an object that delegates to a backing object, providing an alternative interface to it. Because it has no other state, there is no need to create more than one instance of a given adapter to a given object.

Command Pattern

R.PAT.3 Trying to express anything as an operation to be executed, can encourage a imperative style of programming. The `execute()` method of the *Command* pattern is sometimes extended to encompass all kind of executable tasks. An aberrant form of it is the "generic" solution `Object execute( Object o );`
Stick to the principles of object-oriented programming, following the CRC concept: a given class C has a clearly defined set of responsibilities R and collaborates (the second C) with other classes to fulfill its contract.

R.PAT.4 A pattern related to the concept of "command" in the meaning of "executing a function" is the one of "function objects". Function objects (also called functors, a term from predicate logic) are objects whose methods perform operations on other objects, passed explicitly to the methods. Use them judiciously, without trying to emulate functional programming (as in Lisp).
This is especially true when methods return the *this* reference, allowing for operation chaining:

```
anObject.doThis().doThat().andThis();
```

Singleton Pattern

R.PAT.5 Apply the "once and only once" principle, which requires a positive answer to all three questions[8]:

- Will every application use this class *exactly* the same way?
- Will every application *ever* need only *one* instance of this class?
- Should the clients of this class be unaware of the application they are part of?

R.PAT.6 When implementing the *Singleton pattern*, avoid mimicking C++ code in Java. There is no need to check if the instance variable is null (in this respect, the rule #99 in [1] is wrong; this problem is known as "the double-check idiom is broken"[9]).
The following code is much simpler and effective:

```
public class <C> implements Serializable
{
    private static final <C> INSTANCE
        = new <C>();

    public static <C> getInstance()
    {
        return INSTANCE;
    }

    private Object readResolve()
```

[8] See the chapter "When it really is a singleton" in the paper "Use singletons wisely" (URL: http://www-128.ibm.com/developerworks/webservices/library/co-single.html) and the reference to the OnceAndOnlyOnce design principle in the "Resources" chapter of this paper.

[9] The problem of the null check is discussed in depth in the following article, which also refers to other sources of information:
`http://www-106.ibm.com/developerworks/java/library/j-dcl.html`

```
            throws ObjectStreamException
        {
            return INSTANCE;
        }
    }
```

Note the implementation of `readResolve()` to avoid creating new instances during deserialization.

For an in-depth discussion about the Java implementation of a Singleton, consult "When a singleton is not a singleton"[10].

Strategy Pattern, Visitor Pattern

R.PAT.7 Design patterns such as *Strategy* and *Visitor* should be used only with good reasons. *Visitor* is a quite complex pattern, whose implementation raises the issue of accessibility of the visited part. *Strategy* provides algorithmic flexibility, but can lead a functional style of programming. The decision about the choice of the algorithm requires some kind of logic and thus some pre-existent information. The source of it (and their availability) has to be considered during design, but it is also an issue of analysis.
Use a plug-in algorithm when:

- The algorithm can vary from object to object within a class.
- The algorithm is complex enough that it cannot be driven using attribute values alone.
- The algorithm will be different over time.

Flyweight Pattern

R.PAT.8 Apply the *Flyweight pattern* to return a collection of lightweight objects representing a dataset that contains a minimal set of attributes required for browsing. The complete version will be retrieved only if the client makes an explicit decision. A unique combination of attributes (possibly the primary or the candidate key with respect to the underlying database) is used to get the complete object.
Please note that this solution fits seamless with the use of the *DTO pattern* (as known from the J2EE core patterns). The *Data Transfer Object* contains (possibly among other data) the collection of flyweight objects to be presented to the user for selection.

2.3.4. Programming by Contract[11]

Although a formal verification is not possible and even not meaningful in real-life situations, this does not mean that some disciplined approach is not helpful. Thinking about assertions, and generally looking at programming as stipulating

[10] URL: http://java.sun.com/developer/technicalArticles/Programming/singletons/

[11] "Programming by Contract" is explained in depth in the book of B. Meyer, "Object-oriented Software Construction" (Prentice-Hall, 1997, 2nd edition, ISBN 0-13-629155-4).

a contract between client and service provider, helps clarifying basic issues such as required input, delivered results and internal state changes. Even if assertions are simply recorded as comments, they are part of the documentation and help fulfilling the requirements on the software.

R.EXC.1 Consider each method as a contract between the caller and the callee. The contract states the caller must abide by the *precondition*s of a method and the method, in turn, must return results that satisfy the *postcondition*s associated with the method. Additionally, an *invariant* is a condition that must always hold true for an object.

R.EXC.2 Check preconditions, postconditions and invariants by assertions. An assertion[12] is a Boolean expression that must hold true for a program to operate correctly. An assertion validates the assumptions made by a program.

R.EXC.3 Check preconditions at the beginning of a method, and check postconditions at the end of a method before the method returns. Typically, a precondition tests that the object is in a valid state. A postcondition verifies that the object is in a valid state or ensures that the values returned by the method are correct.

R.EXC.4 When developing software, always have assertions enabled, as they will help you catch bugs. Remember that quality assurance testing means running a program with assertions enabled, since it is the job of QA to figure out how to break code.

R.EXC.5 Use the full form of assert, which allows you to implicitly document the assertion by passing an appropriate message to `AssertionError`:

```
assert ( argument != null ) :
        "<class::method> - null return value";
```

The message itself is not internationalized.

R.EXC.6 In `assert`, don't use operations which change the object state through side effects. The assertion should never touch the object itself. It must be possible to run the system with or without assertions, without incurring in unexpected state changes.
If it is necessary to save some data prior to a state change in a postcondition, use two assert statements and an inner class:

```
<METHOD>( final <TYPE> data )
{
   class AssertionData
   {
      private <TYPE> dataCopy;
      AssertionData()
          { // data -> dataCopy as clone }
      boolean isConsistent()
          { // data equals dataCopy }
   }

   AssertionData data;
```

[12] The assertion mechanism is available in Java 1.4 and upward. In earlier versions, it can be simulated by an Assertion class with class-level (`static`) methods.

```
   assert ((data = new AssertionData()) != null );
   …
   assert data.isConsistent();

}
```

R.EXC.7 Correct use of assertions:

- Use assertions to check preconditions and postconditions on parameters of *protected* and *private* access methods (but not in public methods, see later).
- Use assertions to check return values in *public* methods:

  ```
  assert ( result != null ) :
     "<class::method> – null return value";
   return result;
  ```

- Use assertions to check for conditions that shouldn't happen. In this case, use the following idiom:

  ```
  assert false;    // unreachable point
  ```

  If a statement is unreachable as defined in the Java language specification, a compile time error will be emitted. An acceptable alternative is to throw an `AssertionError`.

- Use assertions to check for invalid code branches (e.g. in nested/cascading `if-then-else` constructions or in `switch` statements).

  ```
  if        ( … ) { … }
  else  if ( … ) { … }
  else            { assert false: "…"; }


  switch (<VAR>): …
     default: assert false : <VAR>;
  ```

R.EXC.8 Wrong use of assertions:

- Don't use assertions to do any work. Assertions are developer-level errors and should not be used to repair state in the program or even perform complex operations.
- Don't use assertions to validate parameters of *public* methods. These functions should throw `NullPointerException`, `IllegalArgumentException`, and other relevant exceptions instead. It is up to the responsibility of the caller to provide valid input to a method.
- Don't use assertions to check for user input errors or invalid parameters delivered by other objects.
- Don't bother internationalizing assertion error messages, because they represent developer-level issues, not concerning end users. Always use US-English, the language used for software documentation.

R.EXC.9 In Java 1.4 and upwards[13], use the built-in assertion mechanism. This mechanism can be enabled and disabled as needed, without changing the code.

R.EXC.10 Activation and deactivation of the Java assertion mechanism:

- By default, assertions are off and have to be turned on manually (compiler option). They are disabled by default. Assertions always remain in the code, and can be reactivated by recompiling the code with the option "assertions on".
- The following switches are used (even in combination):
  For the compiler (`javac`):
  - `-source 1.4` (the compiler accept assertions)

  For the runtime (`java`):
  - `-enableassertions` (`-ea`)
  - `-enableassertions:PACKAGE` (`-ea:PACKAGE`)
  - `-enableassertions:CLASS` (`-ea:CLASS`)
  - `-enablesystemassertions` (`-esa`)

  To disable, use `-disableassertions` (`-da`).
- To find out if assertions are on, use this trick:

```
boolean ASSERTIONS = false;
assert  ASSERTIONS = true;
```

  The `assert` expression evaluates and will not thrown an assertion error at any time. If assertions are enabled, then the code will run and will be set to `true` (intentional side effect!). If assertions are disabled, then the code won't be executed and `ASSERTIONS` will remain `false`.
- To enable compilation of program using the keyword 'assert', 'javac' is needed to invoke with flag `-source 1.4`.
  In Eclipse/RAD, go to *Project -> Properties -> Java Compiler -> Compliance and Classfiles* (*Use Configure Workspace Settings*, if using workspace setting) and do the following:
  - Uncheck *Use default compliance settings*
  - Select *Generated .class Files Compatibility* "1.4" and *Source Compatibility* "1.4"
  - Click *Apply*.

  To run the program with assertion enabled, it is needed to specify `-ea` or `-enableassertion` JVM flag.

  In Eclipse/RAD, go to *Run->Run...*

[13] See footnote to R.EXC.2

  - Select the application you want to run in the left pane.
  - Select the argument tab in the main pane.
  - Specify `-ea` or `-enableassertion` as VM arguments.
  - Click *Run*.

- Assertions are only deactivated, not removed. To really strip assertions out of class files, use the following idiom. However, before doing it, ask yourself if a removal is really required and why.

  ```
  static final boolean ASSERTIONS_ON = false;
  if ( ASSERTIONS_ON ) assert <expression>;
  ```

- It is possible to control the default assertion mechanism status for new classes loaded into the JVM by using methods of the `ClassLoader` class.
  The method
  `setDefaultAssertionStatus( boolean enabled )`
  sets the default assertion status, which will be used for classes for which no other instructions exist. This status can be (re-)set to `false` by calling `clearAssertionStatus()`.
  The methods

  -
    ```
    setPackageAssertionStatus
        ( String packageName, boolean enabled)
    ```

  -
    ```
    setClassAssertionStatus
        ( String packageName, boolean enabled)
    ```

  set the status for the named package or class.

  Be aware that the settings will only apply to classes that the class loader has *not* yet loaded into the VM.
  Using

  ```
  static
  {
      ClassLoader.getSystemClassLoader().
          setDefaultAssertionStatus( true );
  }
  ```

  does not provide a real guarantee that all classes will be loaded with assertions enabled.
  To ensure that some code doesn't execute without assertion checking, use the following idiom:

  ```
  static
  {
      boolean assertsEnabled = false;
      assert assertsEnabled = true;
      if ( !assertsEnabled )
         throw new RuntimeException( … );
  }
  ```

2.3.5. Rules for Constructors, Object Creation and Destruction

R.CST.1 Constructor should create fully initialized objects with all of their invariants established.

R.CST.2 There is never a good reason to include a constructor for a class that leaves any of the fields uninitialized, excepted when distinguish between mandatory and optional attributes. In the last case, null can be used to signal a missing value.
Mandatory attributes should always be initialized with appropriate values.
Gratuitous uses of null values inevitably result in `NullPointerException`s.

R.CST.3 Don't assign default values to attributes in their declaration, especially if these values are defined by the Java language specification.

- Assign values to the (private) attributes in the constructor, especially in the case of default values.
- Avoid redundant initializations (guaranteed by the Java compiler) such as:

  -
    ```
    int   j = 0;   OR long   l = 0L;
    float f = 0.0f OR double d = 0.0;
    ```
  - `boolean b = false;`
  - `<CLASS_NAME> object = null;`

R.CST.4 At least mandatory input parameters for constructors must be checked for validity. Whenever possible, all input parameters in constructors should be checked.

R.CST.5 Use the same name for input parameters and their corresponding attributes. Qualify the latter ones with `this`:

```
public class <C>
{
    private <TYPE> <NAME>;

    <C>( <TYPE> <NAME>, … )
    {
       this.<NAME> = <NAME>;
       …
    }
    …
}
```

If this mechanism is not used, carefully select a name which is *really* different from the attribute name, in order to avoid confusions.

R.CST.6 In general, make a defensive copy of each mutable parameter for the constructor (see "Rules for Methods" for further information). In other cases, document the possibility for changes of the original value.

R.CST.7 It may not be always be appropriate to make a defensive copy of a mutable parameter before using it in a constructor. Some constructors require an explicit handoff of the object referenced by the parameter. The constructor that takes control of the client-provided mutable object must make this clear in the documentation. The contract may state that the parameter is in fact immutable, possibly through the use of the `final` modifier.

R.CST.8 If the constructor requires mandatory input parameters and meaningful defaults can not be supplied, suppress the default constructor (make it private). This is also the case if subclassing is not needed. Don't let the compiler automatically create a default constructor, but provide your own and initialize the object to a correct state.
Beware that the JavaBean specification requires a default (empty) constructor. Therefore, in all cases where the object is used as JavaBean, a default constructor is required. If attribute integrity matters, implement a verification mechanism, which checks if an invalid object is being used before complete instantiation (using mutator methods) is done. A simple solution is to use a boolean variable to keep track of the initialization state.

R.CST.9 Constructors must not invoke overridable methods. Never call an abstract method from a constructor in an abstract class.

R.CST.10 `super()` or `this()` must be the first call in a constructor, if such a call is required. This is mandated by the Java language specification.

R.CST.11 If the class implements an interface, assign the object reference obained from the `new` operator to a variable of the interface type, not the class type (e.g. `List list = new ArrayList();`). Doing this decouples the code using the list from the concrete implementation.
However, if the implementation offers specific methods not described by the interface, and such methods can be useful in a given context, the implementation should be used instead. In the case of `ArrayList`, the method `trimToSize()` is not part of the `List` interface. If this method is used, list should be instantiated as `ArrayList list = new ArrayList();`

R.CST.12 In general, creation and reclamation of small objects whose constructors do little explicit work is cheap. They can enhance the clarity and simplicity of a program, enforcing type checking (e.g. "not everything is a string"). But this idiom should not be overused, because the arbitrary creation of a huge amount of objects can adversely influence performance.

R.CST.13 Consider the possibilty of defining a "lightweight object". A lightweight object is an object that does not have a long inheritance chain and does not contain many other objects.

What happens during object construction:

- memory is allocated from the heap to hold all data
- attributes are initialized to their default values
- the constructor of the most derived class is invoked; it calls the constructor of its superclass. The process continues until the constructor for `java.lang.Object` is invoked.

As a consequence, the following situations can lead to performance problems:

- large amounts of code in the constructor
- containment of many objects, or large objects, in initialized as part of the constructor
- deep inheritance hierarchy.

R.CST.14 Don't create a new object when an existing one should be reused. But don't reuse an existing object when a new one should be created (e.g. defensive copy).
If a class has several `set<attribute>(<type> attribute)` methods, consider the possibility of a `reinitialize()` method to reuse the object.

R.CST.15 Eliminate obsolete object references (set them to `null`) whenever a class manages its own memory (e.g. using a collection of objects, adding/removing references from it).

R.CST.16 Avoid building an own object pool. Don't manage memory yourself. This rule applies for every kind of object, including database connections and threads. In the J2EE context, pooling is best done by the container itself – rely on this mechanism.

R.CST.17 Avoid finalizers. Don't use them as destructors, mimicking C++ behavior. Do nothing time-critical in a finalizer, never depend on a finalizer to update critical persistent state. Provide an explicit termination method instead.

R.CST.18 Consider using a factory method instead of a constructor. See the chapter "Applying Design Patterns" for further information.

2.3.6. Rules for Methods

R.MET.1 Choose method names carefully. The names should be self-explanatory and precise.

R.MET.2 Don't provide too many convenience methods.

R.MET.3 For parameter types (both as input to a method and as output from a method), favor interfaces over classes.

R.MET.4 Avoid long parameter lists. Use helper classes (with static methods) to hold aggregates of parameters. This is especially recommended if a frequently occurring sequence of parameters represent some distinct entity. A special case of this situation is the so-called *Data Transfer Object pattern* (see [2]).

R.MET.5 Question the habit of passing `null` to signal the absence of a parameter. Ask for the purpose of such a parameter.

R.MET.6 Use `final` as modifier to the input arguments of a method to signal that the object reference should not be modified.

R.MET.7 Use overloading judiciously, avoid confusion. Don't provide multiple signatures with the same number of parameters, if not really necessary.

R.MET.8 Input parameters should be checked for validity. According to "Programming by Contract", this should *not* be done using assertion, but by conditional checking. In principle, it is up to the responsibility of the caller (client) to provide valid input to a method. However, if for some reasons the called method (callee) has to check for input validity, conditional constructs (`if` statements) must be used.
In public methods, one of the following exceptions must be thrown:

- `IllegalArgumentException`
- `IndexOutOfBoundsException`
- `NullPointerException`.

The exception must be documented using the `@throws` tag of Javadoc. In all cases, checks can be delayed, but must be done before any computation.

Private and package methods must use assertions.

R.MET.9 It is advisable to make a defensive copy of each mutable parameter in a mutator method. Use the `new` operator and not the `clone()` method, because in the latter the type of the parameter can be subclassable (thus possibly containing malicious code).

R.MET.10 Defensive copies are made before checking the validity of parameters. The validity check is performed on the copies rather than on the originals. Be careful: non-zero length arrays are always mutable.

R.MET.11 It could not always be appropriate to make a defensive copy of a mutable parameter before using it in a method. Some methods require an explicit handoff of the object referenced by the parameter. The method that takes control of the client-provided mutable object must make this clear in the documentation.

R.MET.12 Accessor methods should return defensive copies of mutable internal fields. Use `clone()` for this purpose or return an immutable subclass.

R.MET.13 If a method returns an array, never return `null`, but a zero-length array. This special value can be declared as a constant and used in the `return` statement:

```
private final static <TYPE>[] EMPTY_ARRAY
        = new <TYPE>[0];
```

This special case should be appropriately documented as part of the method declaration.

R.MET.14 Delay work that has a good chance of never needing to be done. To meet your performance goals, avoid unnecessary computations. Apply lazy evaluation:

```
<METHOD>()
{
    if ( <ATTRIBUTE> == null )
    {
        // set to valid value
    }

    return <ATTRIBUTE>;
}
```

2.3.7. Methods from java.lang.object and JDK

R.OBJ.1 The `equals()` and `hashCode()` methods must be implemented if the object instances will be stored in a Collection (such as `HashMap`). Otherwise, the behavior (cost function[14]) of this data structure at run time will degenerate to the one of a linear linked list.

equals()

R.OBJ.2 Basic guidelines concerning the `equals()` method:

- No need of such a method if each instance is inherently unique (e.g. `Thread`) or if you don't care (e.g. `Random`)
- No need if the superclass has already an appropriate implementation
- Generally no need if the class is private or has package scope (it will not be used in comparisons)
- Strongly required if object identity is not the same as logical equality
- Equal objects must have equal hash codes
- An object of a derived class that implements `equals()` should not return true when it is compared to its base class for equality. Check carefully the reasons for wanting to support equality comparisons between base and derived class objects.
- Always implement the `equals()` method to accept a parameter of type `Object`, not a derived type.

This method must have the following properties: reflexive, transitive, consistent for all invocations of the same pair of reference values. It must execute deterministic computations on memory-resident objects, not using any external resources.

Skeletal implementation:

[14] The cost function is a function of input prices and output quantity. Its value is the cost of making that output given those input prices. The term is taken from economic theory and used in computing science for analysis of time behavior of data structures and algorithms.

```
public boolean equals( Object o )
{
    if (this == o)              return true;
    if ( !(o instanceof <C>) ) return false;

    <C> instance = (<C>)o;
    …; // checks on attributes of <C>

 }
```

hashCode()

R.OBJ.3 Basic guidelines concerning the `hashCode()` method:

- Important design consideration: a good hash function should produce unequal hash codes for unequal objects. The function should generate an uniform distribution across the whole possible value domain
- Override this method in every class that overrides `equals()`.
- Equal objects must have equal hash codes.
- Include all significant parts of the object in the hash code computation
- Exclude any fields that are not used in equality comparisons
- It is acceptable to exclude redundant fields, that is any field whose value can be computed from fields included in the computation.
- If a class is immutable and the hash code computation is costly, consider caching the hash code in the object rather than recalculating it each time it is requested.
- As alternative, consider pre-calculated hash codes, invalidated by object state changes.
- It is possible to use lazy initialization:

```
private volatile int hashCode;
// guaranteed to be zero by Java language specification

public int hashCode()
{
    if (hashCode == 0 )
    {
        // compute it
    }
    return hashCode;
}
```

- Don't specify a limited domain of value returned by the `hashCode()` method. This would limit the possibility for improvements in future releases.
- Apply the following rules to compute the hash code:

a) start with a constant, nonzero value (at best an odd prime)

b) for every native data type, compute the hash according to the table below

c) if the field is an object reference, and the class provide an `hashCode()` implementation, use it. Otherwise, if the corresponding method is known to be inadequate, recursively invoke `hashCode()` on every of its fields

d) if any field is null, return `0`

e) if the field is an array, compute the hash code for each significant element, applying the above rules

f) at the end, combine the results of the computation and return the value.

Hash computation for a given field `f`:

data type	**hash code**
`boolean`	`(f ? 0 : 1)`
`char, byte, short, int`	`(int) f`
`long`	`(int) (f ^ (f >>> 32))` See footnote [15].
`float`	`Float.floatToIntBits(f)`
`double`	`Double .doubleToIntBits(f)` and then hash the long result using `(int) (f ^ (f >>> 32))` See footnote above.
any object reference `f == null`	`0`

Skeletal implementation:

```
public int hashCode()
{
    int result = 17;
    …
    // for every field do:
```

[15] The >>> operator defines an unsigned right shift. It has the same semantics as the >> operator, except that it always shifts zeros into the high-order bits of the result, regardless of the sign of the left-hand operand. This technique is called "zero extension" and is appropriate when the left operand is being treated as un unsigned value.

```
    result = 37*result + <HASH>;
    …
    return result;
}
```

toString()

R.OBJ.4 Basic guidelines concerning the `toString()` method:

- This method is primarily used for debugging, not for display of information to the end user.
- Should return all of the interesting information contained in the object, in a human-readable form.
- If the format of the string returned is specified, it must be exactly documented.
- Provide methods accessing all of the information contained in the value returned by `toString()`. Don't force the client to parse the string.
- When implementing this method, be careful not to use the `this` keyword by itself as an argument in a string concatenation operation or an `append()` method invocation. Doing so causes `this.toString()` to be invoked, which results in infinite recursion.

compareTo()

R.OBJ.5 Basic guidelines concerning the `compareTo()` method:

- Consider implementing the `Comparable` interface: this allows the objects to be used in conjunction with the utility algorithms of the `java.collection` package, especially for sorting
- Implementing `Comparable` means that the instances of the class have a *natural* ordering. It is up to the responsibility of the designer to provide an implementation which delivers a meaningful order
- The contract for `Comparable` states that this function must satisfy the following properties:

  a) symmetry:
  `sgn( x.compareTo(y)) ==`
  `-sgn(y.compareTo(x))`

  b) transitivity:
  `x.compareTo(y) > 0 && y.compareTo(z) > 0`
  ➔
  `x.compareTo(z) > 0`
  additionally: `x.compareTo(y) == 0` ➔
  `sgn(x.compareTo(z)) ==`
  `sgn(y.compareTo(z))`

  c) equality:
  `(x.compareTo(y) == 0) == x.equals(y)`

- If the argument is not of the appropriate type, the `compareTo()` method should throw a `ClassCastException`
- If the argument is `null`, the `compareTo()` method should throw a `NullPointerException`
- The contract for `compareTo()` specifies only the sign, not the magnitude of the return value. Although the common practice is to use `-1` for "less", `1` for "greater" and `0` for "equal", values other then `1` can be used. One possibility is to build arithmetical differences (e.g. value of attribute 1 - value of attribute 2) in order to signal the possible difference between attributes. This solution is efficient, but care must be taken not to overflow the numerical range (the numerical value of the difference between two integers must be less or equal than $2^{31} - 1$)
- When building the `compareTo()` method, start with the most significant fields and work down the list of the object's attributes. If anything is other than `0`, comparison is finished and the result can be returned.

Skeletal implementation:

```
public int compareTo( Object o )
{
    <C> c = (<C>) o;
    …
    // check fields
    …
    return value; // range ::= {-1,1,0}
}
```

clone()

R.OBJ.6 Basic guidelines concerning the `clone()` method:
Please note that cloning is a complex topic, which should be handled with care.

`clone()` must be implemented only for mutable objects, immutable ones by their very own nature do not require this feature.

- Object cloning is not a simple topic. In general, it is simpler to provide some alternative means of object copying or simply not providing the capability at all
- An elegant solution, avoiding the problem of cloning, is to provide a copy constructor[16]:

  ```
  a) public <C>( <C> sourceObject )

  b) public static <C> getInstance( <C>
     sourceObject )
  ```

[16] Contrarly to some strange assumption floating around, a copy constructor has nothing to do with C++. It was there long before such a language was created, coming from the Smalltalk heritage.

- Implementing object cloning requires a in-depth understanding of shallow and deep cloning, with all consequences. Pay special attention to the following points:

  a) It is required to call `super.clone()` at the very beginning of the `clone()` method implementation

  b) The point above implies catching the `CloneNotSupportedException`

  c) Either the result from `super.clone()` is returned, or further deep cloning operations are performed after it

  d) The `clone()` method should return a regular copy, that is a deep copy (the corresponding method in *Object* returns a shallow copy), in which all references to mutable objects have been cloned.

  e) If *none* of the instance variables references mutable objects, a minimal implementation is possible:

```
public Object clone()
{
   try
   {
      <C> clone = (<C>) super.clone();

      return clone;
   }
   catch( CloneNotSupportedException e )
   {
      throw new InternalError();
   }
}
```

  f) If there are mutable objects, they must be cloned in order to produce a deep copy:

```
public Object clone()
{
   try
   {
      <C> clone = (<C>) super.clone();
      clone.<mutable> =
        (<C>) <mutable>.clone();

      return clone;
   }
   catch( CloneNotSupportedException e )
   {
      throw new InternalError();
   }
}
```

  g) If cloning has to be forbidden along the inheritance line (the reasons should be documented), `clone()` must be declared this way:

```
public final Object clone()
   throws CloneNotSupportedException
{
    throw new CloneNotSupportedException();
}
```

h) Deep cloning in the context of collections means cloning every element of the collection, too.
Example:

```
int size = sourceVector.size();
Vector targetVector = new Vector(size);
for (int i = 0; i < size; i++)
{
    targetVector.add(
       ((<C>)sourceVector.get(i))
       .clone()                    );
}
return targetVector;
```

2.3.8. Hints for Expressions and Control-Flow Structures

Expressions

R.FLW.1 In expressions, clarify the order of operations with parentheses.

R.FLW.2 Order the operands in conditional expressions with AND/OR according to the computational effort and probability of execution.

It is possible to use `&&` and `||` instead of `&` and `|` . In the first case, the second operation will be computed only if required (according to rules of mathematical logic). This conditional evaluation is called "short circuiting" and is applied as follow:

- for `||` (OR operator), the expression is only evaluated if the first operand is *false*
- for `&&` (AND operator), the expression is only evaluated if the first operand is *true*.

Beware of side effects when using these operators.
When the operands are of type integer, the `&` (AND), ^ (XOR) and `|` (OR) operators are bitwise operators.

IF and SWITCH statements

R.FLW.3 When comparing strings, especially in `if` statements, check first for identity and then for equality:

```
// wrong
if ( a.toLowerCase().equals( b.toLowerCase()) )

// correct
if ( a == b || a.equalsIgnoreCase( b ) )
```

R.FLW.4 Consider replacing a long chain of `if-then-else` constructs by a table containing command objects. Using the following idiom, the appropriate action is obtained from the command table and then executed (indirect application of the Command pattern).

```
Map actionMap = new HashMap();
static
{
   actionMap.put(
                  "ACTION_NAME",
                  new Action()
                  {
                     public void action()
                     {
                        // body
                     }
                  }
                );
}
```

R.FLW.5 In a `switch` statement, order the single `case` statements in order of probability of execution (most frequently used cases first).

Loops

R.FLW.6 Label empty statements (e.g. `while (…); //empty`)

R.FLW.7 Code a `break` statement for the last case (even if it is the default case) of a `switch` statement.
But beware that a final `break`❶ statement is unreachable, if a `return` statement follows the `default` statement:

```
switch( … )
{
     case   …: …; break;
     case   …: …; break;
     default : return <VALUE>; break;❶
}
```

This error is generally detected by the compiler, as in the case of compilation in WSAD/RAD.

R.FLW.8 Remember that the `break` statement breaks out of the current loop. If there is an outer loop, it will still run. This is possibly not the intention of the programmer. The same applies to the `continue` statement.

R.FLW.9 Use `break` instead of a boolean variable to check for a premature exit condition from a loop.

R.FLW.10 Beware of hidden goto statements when using `break` und `continue`. Consider the possibility of writing the algorithm using another sequence of statements.
Use labels to clarify the use:

```
RESTART:
{
   for ( … )
   {
       for ( … )
       {
           // body
           break RESTART;
           // body
       }
   }
}
```

A label must be the first statement on a line.
The `break` statement must be nested inside the labeled statement.

```
PROCESS:
{
     for ( … )
     {
          for ( … )
          {
               // body
               continue PROCESS;
               // body
          }
      }
}
```

A label must be the first statement on a line.
The `continue` statement must be nested within the code block of any label to which it refers. A `continue` statement can transfer control only to a label set on a looping construct such as a `for`, `while` or `do` statement.

R.FLW.11 Extract invariants from loops: method calls which deliver the same result during the whole loop execution should be called only once before the loop and the result should be stored in a local variable.

R.FLW.12 Avoid multiple evaluation of initialization code in a loop:

wrong : `for ( int j = 0; j < list.size(); j++ )`

correct:
```
int listLength = list.size();
for ( int j = 0; j < listLength; j++ )
```

R.FLW.13 Don't use an exception to terminate a loop!

```
try
{
   for ( … )
   {
      // BODY
   }
}
catch( IndexOutOfBoundsException e )
{
   // ignore
}
```

R.FLW.14 When manipulating static variables, especially in a loop, use a temporary variable of the same type (access to class attributes is more costly that access to instance attributes):

```
<T> j = staticVariable;
// manipulate j
staticVariable = j;
```

Other

R.FLW.15 Use a `finally` block to release resources.

2.3.9. Best Practices in Exception Handling

General rules

R.CSY.1 Exceptions are intended to be used in situations that happen rarely. In particular, they should not be part of the normal flow of an application. Don't misuse exceptions for every error condition. An exception is raised only when exceptional conditions are met. Never use exceptions for control flow.

R.CSY.2 A method should only throw an exception to indicate some sort of failure condition. Under normal circumstances, when provided with valid arguments and when used services are functioning normally, a method should not throw an exception. The corollary to this is that a method should do what is expected of it *or* throw an exception. It is expensive for the caller to have to check a return code which nearly always indicates success, and the checking and handling code obfuscates the primary logic flow.

R.CSY.3 A method may sometimes avoid the throwing of exceptions by called methods (or within the method itself) by checking preconditions. This should be done if and only if the exception would be thrown often, or the check represents a reasonable variation on the normal, expected logic flow.

R.CSY.4 Use assertions to catch logical errors in the code. Consult the chapter about programming by contract in this document for further information about assertions and their use.

R.CSY.5 Use other mechanisms to report expected state changes: return codes, sentinel values, methods returning a status (as value or as object).

R.CSY.6 Signaling exceptional conditions by returning null values is not a good idea. Doing so limits control flow to the ordinary paths of method invocation and return, and buries evidence of where an exceptional condition occurred.

R.CSY.7 Catch exceptions at the point closest to the occurrence of the exception that allows execution to proceed in a stable way. Raising and catching an exception is generally expensive while arranging to catch one (with a try/catch block) is cheap.

R.CSY.8 In a layered architecture, the exceptions of the lower layer will need to be caught and handled, or else they must be converted into exceptions applicable in the higher layer.

Implementation rules

R.CSY.9 Never declare `throws Exception`, always use a subclass of `Exception`.

R.CSY.10 It could possibly make sense to separate the fatal error conditions from the recoverable ones. In order to avoid a lot of exception classes, reduce the overall number of exceptions by categorizing them and using additional information (e.g. typesafe constants in an enumeration to represent the error condition).

```
public class Recoverable<X>Exception
  extends Exception
{
  public Recoverable<X>Exception
      ( String reason, <CODE> code )
  {
      super( reason );
      this.code = code;
  }

  public final <CODE> getReasonCode()
  {
       return code;
  }

  private <CODE> code;

 }
```

➔
```
try { … }
catch(Recoverable<X>Exception e )
{
     <CODE> code = e.getReasonCode();
     // act appropriately, according to code
}
```

R.CSY.11 Avoid using `try-catch` on a per method basis for all methods within a block.

R.CSY.12 Never let exceptions propagate out of a `finally` block. This includes the `NullPointerException`.

R.CSY.13 Ensure that an exception is never propagated out of a `static` block.

R.CSY.14 If an exception is thrown in a constructor, the `new` operator does not return. Instead, control transfers to the matching exception handler.

R.CSY.15 Use unchecked, run-time exceptions to report serious unexpected errors. Such errors indicate flaws in the programming logic and should be spotted by systematic testing. If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception[17].

R.CSY.16 Provide as much information as possible to help tracing the error.

R.CSY.17 There is seldom any purpose in translating an error message into a specific language until the message is actually displayed to a human. If this is the case, use a key to a resource bundle in the text passed to the exception. The key should have a predefined format

[17] See http://java.sun.com/docs/books/tutorial/essential/exceptions/runtime.html about Exception and RuntimeException.

and should contain class name, method name and a symbol which shortly identifies the exception cause.

R.CSY.18 Only convert exceptions to add information, not to suppress or change it.

R.CSY.19 Do not use the expensive `instanceof` operator to avoid `ClassCastException`. It is poor practice to use such a test to eliminate an unlikely casting exception:

```
<CLASS> class;
try
{
     class = (<CLASS>) object;
}
catch( ClassCastException e )
{
     …;
}
```

is preferred over:

```
<CLASS> class;
if ( object instanceof <CLASS> )
{
     class = (<CLASS>) object;
}
else
{
     …;
}
```

unless `o` is likely *not* to be of class `C`.

R.CSY.20 Do not silently absorb a run-time or error exception (e.g. `ArrayOutOfBoundsException`). In general, never ignore an exception. One possible exception to this rule is an interrupt (`InterruptedException`[18]).
If it is not clear what to do in such an exceptional case, the method `printStackTrace()` must the used at least.

R.CSY.21 Return objects to a valid state before throwing an exception.

R.CSY.22 The reuse of exceptions is not really useful. The information about the stack trace is not available, because it is generated when the exception is thrown at execution time.
Example:

```
private final Exception EXCEPTION =
        new <NAME>Exception(…);
…
try
{
     …
     throw EXCEPTION;
```

[18] Thrown when a thread is waiting, sleeping, or otherwise paused for a long time and another thread interrupts it using the interrupt method in class `Thread`.

```
}
catch( … ) { … }
```

2.4. Performance and Optimization

The first step to fast code is properly designed code. Choose the proper algorithms and the data structures most suited to the problem.

Without clear performance objectives, tuning will never be completed.

Before touching any code:

- Ask yourself exactly what the application is doing and why. Then question whether it needs to do in that way, or even if it needs to do it at all.
- Establish and write down meaningful and reasonable performance requirements with the stakeholders.
- Base your design on abstractions and interfaces, which allows to change the underlying implementation without modifying the whole system.
- Apply strategies that do not require changing the source code (e.g. initial heap size of the Java VM).
- Use tools for perfomance analysis, profiling and code path coverage.
- Measure and verify! Profile the code and look at methods that consume most of the CPU time, and excessive short-lived object creation.
- Consider multi-threaded behavior.
- Look at external resource usage and operations which are generally known as being expensive (in terms of execution time or memory space).

Do not optimize code unless you have proved that optimization is necessary. After performing an optimization, profile again to prove that it has the desired effect. Each set of changes should be followed by a run of benchmarks to precisely identify improvements (or degradations!) in the performance across all functions.

Consider the following techniques to improve the code:

1. Stepwise replacement

   Identify the bottlenecks in the code, and replace code fragments, methods or whole class implementation by more performant versions. Introduce small changes at once, and measure the performance before and after the replacement.

2. Use of specific solutions offered by the Java JDK (see also the chapter "Solutions to Specific Problems, Use of Libraries").

2.1. Use buffered I/O

```
PrintWriter writer = new PrintWriter(
      new BufferedWriter( new FileWriter( … ) ) );
```

2.2. If there is a need to represent data with many boolean value (e.g. a statistical survey of market data), use `java.util.BitSet`. Although `BitSet` operations are slower than assignments to `boolean` variables, a tiny `BitSet` with a single-element array can represent 64 states (because it uses internally an array of `long`, and a `long` is 64 bits wide) and is usually very efficient.

2.3. Don't use `String` or `StringBuffer` for any task that involves looping over the characters of a string. Instead, use character arrays directly. Obtain the array using the `toCharArray()` method, and after processing convert the array back to `String` using the methods `valueOf(char[] data)` or `copyValueOf(char[] data)` in the `String` class.

2.4. Don't use for the + operation for string concatenation, use `StringBuffer.append()` instead. See chapter "String Handling" for further informations on strings.

2.5. Generally, the JDK methods that convert objects and data types to strings are suboptimal. Alternative implementations can be considered[19].

2.6. Local (temporary) variables and variables used as arguments to methods are the fastest way to access variables and update them.

2.7. Prefer `int` over `long`, prefer `float` over `double` (if the range is sufficient).

2.8. In general, comparison to `0` is faster than comparison to other numerical values.

2.9. Use `System.arraycopy()` to copy arrays.

2.10. Array-element access is generally slower than accessing atomic variables (probably due to the range checking for the elements and the null reference checking for the array itself done by the Java VM).
Use local variables with reference to array elements for multiple access within a loop.

2.11. Avoid casts whenever possible.

2.12. Pay attention to:

- Code that is in a loop.
- Object creation, especially in frequently used methods.
- Code using collections.

[19] If a replacement is considered, the request should be submitted for approval to UBS Solution Life Cycle. It could be then be integrated into the Common Lib suite of public available libraries.

Use the techniques described in this chapter to optimize the code.

3. The following techniques, used in compiler construction, can be applied to optimize the code. Use them with care, as generally this will be done by an optimizing compiler.

   3.1. Empty method removal

   Identify methods with empty method body and remove them.

   3.2. Dead code removal

   Remove code which is never used and never reached by the control flow.

   3.3. Strength reduction

   Replace an expensive operation with a more efficient one.

   3.4. Constant folding

   Not recommended, already done by all compilers.

   3.5. Common expression elimination

   Use temporary variables to replace common parts of expressions.

   3.6. Loop unrolling

   Not recommended, some compilers do it based on internal heuristics.

   3.7. Algebraic simplification

   Use algebraic transformations to make expressions simpler.
   One application to logical operators is the so-called "short circuiting" for `&&` and `||` operators.

   3.8. Loop invariant code motion

   Move operations which do not change during loop execution outside the loop.
   The most common error is to call a method in the header of the `for` loop. The expression will be evaluated as many times as the loop is executed:

```
String s =…;
for (int i = 0; i < s.length(); i++ ) { … }
```

   instead use:

```
String s =…;
int l = s.length();
for (int i = 0; i < l; i++ ) { … }
```

Adjusting the heap of the Java VM:

- In order to estimate the appropriate initial value, create a background thread that periodically checks the values from `totalMemory()` and `freeMemory()` in the `Runtime` class. The approximated results help estimate the starting values.
  Use this simple and quick solution as starting point, but then switch to tools specific for this purpose.

- The thread stack can be reduced if the application doesn't use threads. A default of 1MB is large, in such a case 256K can be used instead.

3. Use of Specific Java Mechanisms

3.1.1. Concurrency and Synchronization

The topic of concurrent programming is difficult and requires in-depth understanding of all issues related to it, down to operating-system concepts. Don't apply these mechanisms if technical knowledge and practical experience in this field is missing.
The literature (see [C-1], [C-2], [C-3], [C-4]) gives a first introduction to this topic in Java.

Carefully analyze the requirements which lead to the use of concurrency mechanisms and find the rationale behind this choice.

R.CSY.1 Thread management is **forbidden** in J2EE.

R.CSY.2 Don't write own synchronization mechanisms, garbage collectors, memory management pools, daemons, schedulers.

☞ Java 5 delivers in `java.util` the `concurrent`, `concurrent.atomic`, and `concurrent.locks` packages. According to Sun, they "provide a powerful, extensible framework of high-performance, scalable, thread-safe building blocks for developing concurrent classes and applications, including thread pools, thread-safe collections, semaphores, a task scheduling framework, task synchronization utilities, atomic variables, and locks."

If threads are used in standalone applications (which require formal approval by UBS TA), at least the following *minimal* rules apply:

R.CSY.3 Threads are not a "silver bullet" for improving application performance. There is an overhead required to switch between threads. First verify the design, and measure the behavior of the application using performance analyzers. Based on the result, decide if threads are really required.

R.CSY.4 Use threads if the application needs:

- to react to many events simultaneously.
- to provide a high level of responsiveness.
- to take advantage of machines with multiple processors.

R.CSY.5 Avoid *unnecessary* synchronization, check requirements in this respect.

R.CSY.6 Use synchronized wrappers to provide synchronized interfaces. For example, the Java Collections offer both synchronized and unsynchronized versions of data structures.

R.CSY.7 Do not synchronize an entire method if the method contains significant operations that do not need synchronization.

R.CSY.8 When writing code which is required to be thread-safe, the following operations will require a synchronization:

- Because a (memory) word in Java is 4 bytes, `long` and `double` are the only two types comprised of two words. For these types, a memory read or write is *not* atomic.
- The same is true for the increment (`++`) and decrement (`--`) operators, which are not atomic. They represent a consecutive read and write operation.

3.1.2. Serialization

R.CSY.9 Consider using `Externalizable` rather then `Serializable`, especially for complex object structures involving the serialization of deep object graphs or if performance is the primary concern. Serialization can be improved by turning offf the standard serialization format. By declaring the class `Externalizable` instead of `Serializable`, and implementing the two methods `readExternal()` and `writeExternal()`, one can control the serialization process by manipulating the streams (input, output) directly.

```
public void readExternal( ObjectInput in )
{
  <ATTRIBUTE> = in.read<TYPE>();
  <ATTRIBUTE> = (<TYPE>) in.readObject();
}

public void writeExternal( ObjectOutput out )
{
  out.write<TYPE>( <ATTRIBUTE> );
  out.writeObject( <ATTRIBUTE> );
}
```

For strings, the methods `readUTF()` and `writeUTF()` are used.

4. Solutions to Specific Problems, Use of Libraries

Use libraries, don't waste your time writing ad hoc solutions to problems only marginally related to your work!

4.1.1. Data Structure Implementation, Collections

There is no universal best choice for data structures. The proper choice depends in the operations performed on it, and how frequente there are.
The following hints can be helpful. For further information, please consult the literature reference at the beginning of the document (item [A-1]).

- ☐ `LinkedList` or `ArrayList` should be used for collecting elements for sequential iteration in index order. Duplicates are allowed.
- ☐ `LinkedList` should be used for queues, double-ended queues ("deques"), and stacks
- ☐ `ArrayList` should be used for random access or access by index.
- ☐ `HashSet` and `HashMap` should be used when random access by element (set) or key (map) is needed, but no sequential access in element/key order.
- ☐ `LinkedHashSet` and `LinkedHashMap` support random access, but sequential access (iteration using an `Iterator`) in element/key order is also effective.

See the appendix "Data Structures" for more information on runtime behavior.

General Issues

R.COL.1 If data structures are needed, consider the ones offered by the `Collection` package as the first choice. Avoid implementing own data structures.

R.COL.2 This package offers different algorithms, which are orthogonal[20] to the data structures, e.g. sort[21], reverse, shuffle.

Example:

```
   Collections.sort( list,
                     String.CASE_INSENSITIVE_ORDER );
```

[20] "Orthogonal" (a term taken from geometry) means in this context that the algorithm can operate on any type of collection, that is, it is not bound to a single, specific collection.

[21] The sorting algorithms in the `Arrays` and `Collections` classes are stable, that is, they do not reorder equal elements.

R.COL.3 Declare references and return types to be of the `Collection` interface types.
Examples:

```
      Set set   = new HashSet();
      List list = new LinkeList();
      Map  map  = new HashMap();
```

R.COL.4 Understand the difference between synchronized and unsynchronized versions of the collections. Use them appropriately.

R.COL.5 Collections offer a read-only version, too. If the data structure is filled once, and only accessed, but never changed afterwards, use the read-only version.
A known-sized array is the fastest way in Java to store and access elements of a collection.

R.COL.6 Try to pre-size any collection to be as big as it will need to be. It is better for the object to be slightly bigger than necessary than to be smaller than it needs to be.

R.COL.7 There is a strong relationship between the `compareTo()` method in `Comparable` and `Comparator` at one side, and the `Collections` at the other side. A comparator is used primarily to override default ordering and therefore is not a method normally implemented. A partial[22] ordering comparator is used with the `sort()` methods of `Collections` and `Arrays`. A total[23] ordering comparator is used in the constructor of a class that implements the `SortedSet` or `SortedMap` interfaces, such as `TreeSet` or `TreeMap`.

R.COL.8 A `Comparator` implements the logic for comparisons between pair of elements in a collection:

```
public class BackwardOrder implements Comparator
{
   public int compare( Object first,
                       Object second )
   {
      Comparable firstComparable =
                 (Comparable) first;
      Comparable secondComparable =
                 (Comparable) second;
      return secondComparable
               .compareTo( firstComparable );
   }
}
```

➔ `Collection.sort( list, new BackwardOrder() );`

Instead of creating every time a new element, a functional object can be used:

```
public static Comparator BACKWARD_ORDER
   = new Comparator()
{
   public int compare( Object first,
                       Object second )
```

[22] In partial ordering, a comparator does not use all of the instance variables for comparison.

[23] In total ordering, the same set of instance variables is used in `equals()` and in `compareTo()`.

```
   {
      Comparable firstComparable =
                 (Comparable) first;
      Comparable secondComparable =
                 (Comparable) second;
      return secondComparable
               .compareTo( firstComparable );
   }
};
```

➔ `Collection.sort( list, BACKWARD_ORDER );`

R.COL.9 The following bulk operations in the `Collection` interface modify the collection, resulting in a collection different from the original one.

- `addAll( Collection c )`
  Adds all of the elements in the specified collection to this collection.
- `retainAll( Collection c )`
  Retains only the elements in this collection that are contained in the specified collection.
- `removeAll( Collection c)`
  Removes all this collection's elements that are also contained in the specified collection.

The boolean result type indicates if the collection was modified as result of executing the bulk operation.

R.COL.10 Combining operations on the containers in the `Collections` framework can simplify the code.
Examples:

- does the first map contain all of the elements in the second map?

```
boolean isSubMap = first
           .entrySet()
           .containsAll( second.entrySet() );
```

- initialize a list to default values:

```
List list =
   new ArrayList( Collections
                    .nCopies( COUNT_VALUE ) );
```

- iterate backward through a list:

```
for ( ListIterator iterator
        = list.listIterator( list.size();
      iterator.hasPrevious()             ; )
{
    // get element with iterator.previous();
}
```

R.COL.11 When iterating over a collection, consider the different possibilities:

- `Iterator`
- `Enumeration`

- access method in a loop (e.g. `get(int i)` in `Vector`).

  The last possibility is the fastest one for lists:

```
 // assume <CLASS> element
int size = list.size();
for ( int j = 0; j < size; j++ )
{
    element = (<CLASS>) list.get( j );
}
```

Operations on arrays

R.COL.12 The `java.util.Arrays` package offers operations for arrays of built-in types (excepted for `boolean`), such as `sort()` and `binarySearch()`.
For arrays of objects, these methods can be applied, but require that the corresponding classes implement `Comparable` or at least that a `Comparator` is passed to the method.

Map collections: HashMap, HashSet

R.COL.13 In the current implementation of the Collection package, `HashSet` and `HashMap` have a very similar implementation. In fact, the set implementation uses a map with a dummy value for each entry (set element). The same is true for `TreeSet` and `TreeMap`.

R.COL.14 Use a `TreeSet` or `TreeMap` when an ordered iteration is required. It is possible to create an ordered iterator from an unsorted `HashSet` as follows:

```
Iterator iter = new TreeSet( set ).iterator();
```

But if no ordering is required, use `HashSet` instead, because it is much faster. Similarly, use `TreeMap` when ordering is required, and `HashMap` otherwise.
`LinkedHashSet` should be used only if order matters or it is required to iterate fast over all elements.

R.COL.15 Use a `Set` (implemented as `HashSet`) to eliminate duplicates from a collection.

```
Collection list = new LinkedList();
list.add(…);
…
Collection set = new HashSet( list );
```

R.COL.16 The `add()` method in `HashSet` returns a boolean. The value `true` signals that the element was added to the set, the value `false` means that the element is already present.

R.COL.17 All of the general-purpose implementations of the `Map` interface allow `null` as both a key and value, which means that a `null` return value from `get(),put(),remove()` could mean any-thing.
To be completely sure, the method `containsKey()` must be invoked beforehand to determine if the key exists:

```
if ( map.containsKey( key ) )
{
      map.remove( key );
}
```

R.COL.18 Use the following idiom to iterate over all elements of a `Map`:

```
Iterator iterator = map.entrySet().iterator();
while ( iterator.hasNext() )
{
      Map.Entry element
         = (Map.Entry) iterator.next();
      Object key   = element.getKey();
      Object value = element.getValue();
}
```

R.COL.19 Sets are not ordered (as in mathematical set theory), but implementations of the `SortedSet` interface, such as `TreeSet`, maintain their elements in ascending sorted order. To this purpose, the elements must implement the `Comparable` interface or a `Comparator` must be used when creating the sorted set.
If the container is required to maintain the order in which elements are added, one of the general-purpose `List` implementations should be used.

R.COL.20 A `List` and a `Set` cannot be equal even if they contain exactly the same elements, because sets are unordered. Maps are not equal if any of the keys map to different values, even if they contain exactly the same keys.
To known if two maps have the same set of keys, use
`a.keySet().equals( b.keySet() )`
The same can be done using the `values()` method.

R.COL.21 Use of hash tables in the Collections framework (`HashMap`, `HashSet`):

- the smaller the table is, the fewer the index values, the more likely a hash collision
- the larger the table is, the more memory is wasted and the longer it takes to iterate over a collection view of the map.

R.COL.22 Use higher capacity and lower load factors in `HashMap`. This will increase search performance, at the expense of increased space.
Guidelines:

- The smaller the table is, the fewer the index values, the more likely a hash collision.
- The larger the table is, the more memory is wasted and the longer it takes to iterate over a collection view of the map
- `initialCapacity` must be at least greater than the number of elements.
- `initialCapacity` must be kept to a minimum.
- Excess capacity is bad for iterators, because the iterator iterates over all the nodes in the map.

- If the default load factor LF[24] is used (default value is `0.75`), but an initial capacity is required, pick a number that is about twice the size of the container is expected to grow to. If possible, use a prime number near to this number. The default load factor is good, but only when using two times the expected size of the hash table as the initial capacity.
- When using an exact initial size (e.g. for fixed lookup tables), set the load factor to `1.0` .

R.COL.23 When using strings as keys in a hash table, use a specific class instead of the general purpose `String` class. The class itself is smaller in terms of memory and the retrieval operations will be faster, because the hash code is already computed.

```
public final class Key
{
  private final String key;

  public Key( String key )
  {
     this.key = key;
     hash     = key.hashCode();
  }

  public final int hashCode()
  {
     return hash;
  }

  public final boolean equals( Object o )
  {
     if ( this == o ) return true;
     if ( o == null ||
          getClass() != o.getClass() )
        return false;
     Key other = (Key) o;

     return key.equals( other.key );
}
```

If the string space of the keys is really stable, consider using the `intern()` method of the `String` class:

```
  public Key( String key )
  {
     this.key = key.intern();
     hash     = key.hashCode();
  }
```

In this case, use an equality comparison in `equals()`:

```
     return key == other.key;
```

R.COL.24 How to implement set operations (in the mathematical sense) using `Set`:

[24] The load factor LF is used to rehash the table when LF% of the total capacity of reached.

```
Set firstSet  = new HashSet();
Set secondSet = new HashSet();
// … add elements to the two sets …

// subset of a set
boolean isSubset =
  secondSet.containsAll( firstSet );

// union of two sets
Set union = new HashSet( firstSet );
union.addAll( secondSet );

// difference of two sets
Set difference = new HashSet( firstSet );
difference.retainAll( secondSet );

// symmetrical difference of two sets
Set result = new HashSet( firstSet );
result.addAll( secondSet );
Set temp = new HashSet( secondSet );
temp.retainAll( secondSet );
result.removeAll( temp );
```

➔ an element is either in the first set or in the second set, but not in both

R.COL.25 Set operations can be also realized on `Map`:

- union: `a.putAll( b );`
- intersection: `a.keySet() .retainAll( b.keySet() );`
- symmetric difference: `a.keySet() .removeAll( b.keySet() );`
- submap: `a.keySet() .containsAll( b.keySet() );`

R.COL.26 Use `WeakReference`s to hold elements in (generally) large canonical lookup tables. An entry is automatically deleted if no reference to the corresponding key exists anymore.

R.COL.27 Use `SoftReference`s for cache elements. `LinkedHashMap` can also be used to implement a cache. In this case, the method `removeEldestEntry()` must be overwritten.

List collections: LinkedList, ArrayList

R.COL.28 If the order of the elements is revelant, use a `List.` Such a collection keeps the elements in the order they are inserted (using `add()`).

R.COL.29 Use `ArrayList` for random access, use `LinkedList` when adding or removing elements from or to the beginning or the middle of the list.

R.COL.30 It is possible to create a fixed-length `ArrayList` that outperforms the general-purpose `ArrayList` implementation:

```
List list = Array.asList( List.toArray() );
```

The operartion itself is expensive, but further operations on this fixed-length list are faster than on the general implementation.

R.COL.31 A `List` can be converted to an array using the `toArray()` method. The class `Array` has a method `asList( Object[] array )` which generates a List from the given array, still using internally the array itself. All methods of the class `List` can then be used by the array, excepted for `add()`, `addAll()`, `remove()`, `clear()`.
When sorting an array, the `sort()` method of the `Arrays` class can be called directly. A (sorted) list can be generated later, if needed:

```
String s[] = { … };
Arrays.sort( s );
List list = Arrays.asList( s );
```

R.COL.32 Lists are not equal unless the elements are in the same order.

R.COL.33 The default initial size is `10` for `ArrayList`.

- o Use `trimToSize()` only if the list has considerably shrunk.
- o Invoke `ensurecapacity( int minimum )` before adding a lot of elements at one time.
- o An alternative is to enlarge the list using dummy elements as placeholders:

```
list.addAll( Collections
                   .nCopies( 1000, null ) );
```

R.COL.34 It is permissible for lists to contain themselves as elements, but a `Set` cannot contain itself. However, `equals()` and `hashCode()` are no longer well-defined on such lists.

R.COL.35 Use the `List` object returned by `sublist()` only as a transient object. Otherwise, the semantics of this list could become undefined if elements are added to or removed from the underlying list.

R.COL.36 The `java.util.Stack` is a legacy implementation that subclasses the synchronized `Vector` class. A `LinkedList` can also be used to implement a stack, using `addLast()` and `removeLast()`. Similarly, a queue can be implemented with `addLast()` and `removeFirst()`.

4.1.2. Date and Time

Please note that "date" and "time" are very complex topics, as all concepts related to calendrical calculations. The representations used in programming languages and libraries are approximations, which are useful for daily business, but possibly not correct in astronomical and geophysical terms.

R.DTM.1 The method `System.currentTimeMillis()` returns a long (64-bit integer), representing the elapsed time in milliseconds from January, 1th 1970 (the so-called Epoch[25]). The same representation is used in `java.util.Date`. The only purpose of this class is now to store a long and pass it around without modifications.

R.DTM.2 There is a mismatch between `java.util.Date` and `java.sql.Timestamp`: the latter is the only class in JDK which has a time representation down to the nanoseconds[26].
In order to retain (at least *partially and approximately*) the nanosecond part of an SQL timestamp, the following solution[27] can be applied.

```
java.sql.Timestamp ts = …;
Date d = new Date( ts.getTime() +
                   ts.getNanos()/1000000 )
```

R.DTM.3 The `java.util.Date` class should not be used directly (nearly all of its methods are deprecated). Use `GregorianCalendar` instead. Note that this class implements the interface `Calendar`, allowing you to use the known idiom:

```
      Calendar c = new GregorianCalendar();
```
or
```
      Calendar c = Calendar.getInstance();
```

R.DTM.4 If only a specific functionality is used, `java.util.Date` or `java.util.Calendar` can be replaced by a simple class tailored to a specific usage

The following example[28] is one possible solution for a dedicated class with limited functionality:

```
import java.util.*;
import java.text.*;

public final class SimpleDate
{
     private static Date date  = new Date();
     private static DateFormat formatter =
               DateFormat.getDateInstance();

     public static String format()
     {
        date.setTime( System.currentTimeMillis() );
        return formatter.format( date );
     }

     public static String format( long milliseconds )
     {
        date.setTime( milliseconds );
        return formatter.format( date );
     }
}
```

[25] The Unix epoch is the time 1970-01-01T00:00:00Z (in ISO 8601 format).

[26] 1 nanosecond corresponds to 10^{-9} seconds.

[27] It follows the equivalence `1 millisecond = 1 *` 10^6 `nanos`.

[28] A similar solution can be derived for time and timestamp.

Reject negative arguments if only dates after the Epoch should be represented:

```
 if ( milliseconds < 0 )
          throw new IllegalArgumentException(…);
```

R.DTM.5 The class `GregorianCalendar` represents a moment in time calculated to the nearest millisecond.

Each constructor in `GregorianCalendar` requires date information in addition to time information. It is not possible to represent time without a date in this class. If part of the information is missing, the corresponding value is set to zero. The constructor does not accept an argument for milliseconds. Such a value must be added after construction using

```
  calendar.set( Calendar.MILLISECOND, value );
```

This method can be used in conjunction with the `SQLTimestamp` to add nanonseconds to the date value, as explained previously for `Date` in this chapter.
The factory method `Calendar.getInstance()` generates a `Calendar` implementation using the calendar system valid for the platform where the Java system is running. Because Sun didn't implement any other calendar except for the Gregorian one[29], the two calls

```
      Calendar.getInstance()
```

and

```
      new GregorianCalendar()
```

are equivalent in practical terms.

R.DTM.6 Recipes for the Calendar API:

- calculate elapsed time between dates in milliseconds:

```
Calendar c1 = new GregorianCalendar(…);
Calendar c2 = new GregorianCalendar(…);
long delta  = c2.getTimeInMillis() -
              c1.getTimeInMillis()  ;

// how to obtain time units
// as elapsed time (how many days/hours/…)

long seconds = delta / 1000;
long millis  = delta % 1000;
long hours   = seconds / 3600;
seconds      = seconds - (hours*3600);
long minutes = seconds / 60;

seconds      = seconds - (minutes*60);
long days    = hours / 24;
hours        = hours % 24;

// day calculation - approximated

long days = Math.round( (double) delta /
```

[29] At least as of January, 2006. No plans for other implementations are known until yet.

```
                    24. / 60. / 60. / 1000. );

    // day calculation - exact

    long days;
    c1.clear( Calendar.MILLISECOND );
    c1.clear( Calendar.SECOND );
    c1.clear( Calendar.MINUTE );
    c1.clear( Calendar.HOUR_OF_DAY );
    c2.clear( Calendar.MILLISECOND );
    c2.clear( Calendar.SECOND );
    c2.clear( Calendar.MINUTE );
    c2.clear( Calendar.HOUR_OF_DAY );

    while ( c1.before( c2 ) )
    {
        c1.add( Calendar.DATE, 1 );
        days++;
    }

    // month calculation - exact

    long months;
    c1.clear( Calendar.MILLISECOND );
    c1.clear( Calendar.SECOND );
    c1.clear( Calendar.MINUTE );
    c1.clear( Calendar.HOUR_OF_DAY );
    c1.clear( Calendar.DATE );
    c2.clear( Calendar.MILLISECOND );
    c2.clear( Calendar.SECOND );
    c2.clear( Calendar.MINUTE );
    c2.clear( Calendar.HOUR_OF_DAY );
    c2.clear( Calendar.DATE );

    while ( c1.before( c2 ) )
    {
        c1.add( Calendar.MONTH, 1 );
        days++;
    }
```

- determining a person's age:

```
Calendar birthday = …;
Calendar today = Calendar.getInstance();
int age =     today.get( Calendar.YEAR ) -
          birthDate.get( Calendar.YEAR );
birthday.add( Calendar.YEAR, age );
if ( today.before( birthDate ) )
{
   age--;
}
```

- compute the time offset from UTC:

```
int offset = calendar
               .get( Calendar.ZONE_OFFSET );
offset = offset / 60000;

// offset can be positive or negative

int minutes = offset % 60;
int hours   = offset / 60;
```

```
// give offset as delta (no deplacement)

int minutes = Math.abs( offset % 60 );
int hours   = Math.abs( offset / 60 );
```

- given local time, find time in other timezone:

```
Calendar local = …;
Calendar other = new GregorianCalendar(
   TimeZone.getTimeZone( "<TIMEZONE>" ) );
other
   .setTimeInMillis( local.getTimeInMillis() );

// "other" contains time in another zone
```

- given time in other timezone, find local time:

```
Calendar local = …;
local
   .setTimeInMillis( other.getTimeInMillis() );

// "local" contains local time
```

- to add or substract day/month/year to a calendar date, use `add()` or `roll()`.
  For both, a field of Calendar is specified. A positive value is added, a negative one is substracted. In `roll()`, the operation only affects the specified field, while `add()` adjusts other fields if needed (the next larger field is modified when the result makes the calendar "rolls over". Take care for inconsistencies due to the semantics of the operation, which is not correctly reflected by the name:

```
// before changes: date is January, 20 1999
calendar.roll( Calendar.MONTH, false );
// date is now December, 20 1999 - wrong !

// before changes: date is January, 20 1999
calendar.add( Calendar.MONTH, -1 );
// date is now December, 20 1998 - correct
```

- compute the number of days in a month:

```
calendar
  .getActualMaximum( Calendar.DAY_OF_MONTH );
```

- find the day in the week:

```
int weekDay = calendar
                .get( Calendar.DAY_OF_WEEK );
// 1 = Sunday
// no common agreement on "beginning of week"
```

- the idiom mentioned above for the day in the week can be applied to other date units, such as

```
Calendar.DAY_OF_MONTH,
Calendar.DAY_OF_WEEK_IN_MONTH,
Calendar.DAY_OF_YEAR
```

R.DTM.7 Computing elapsed wallclock time:

```
long start   = System.currentTimeInMillis();
long elapsed = System.currentTimeInMillis() -
               start;
double seconds = elapsed / 1000.;
double minutes = elapsed /      (60.*1000.);
double hours   = elapsed /      (60.*60.*1000.);
double days    = elapsed /  (24.*60.*60.*1000.);

// using double allows to store the fractional
// part of the elapsed time unit
```

4.1.3. String Handling

R.STR.1 Avoid strings when other types are more appropriate: strings are poor substitutes for value types, enumerated types, aggregate types (with multiple components).

R.STR.2 Avoid explicit object creation for string *constants*:
wrong: `String s = new String( "…" );`
ok: `String s = "…";`

R.STR.3 Use the concatenation operator for string *constants* in declarations:
`String s = "…" + "…" + "…";`
But *never* use the concatenation operator otherwise.

R.STR.4 Remember that the empty string is represented by `""`, not by `null`.

R.STR.5 Avoid using `StringTokenizer`, because it is not very efficient. In Java 1.4 and higher, use the `split()` method of `String` instead. It accepts a regular expression (see the chapter "Regular Expressions") as delimiter for the splitting process (e.g. `split("\\s")` separates on blanks). An overloaded version takes an additional parameter of type `int` (called "limit") which specifies the number of times the pattern is applied.

R.STR.6 There is no `join()` method pairing with the `split()` method mentioned above, but it can be easily implemented:

```
public static String join( String[] stringList,
                           String separator )
{
   StringBuffer buffer = new StringBuffer();
   int stringCount = stringList.length - 1;

        for (int j = 0; j < stringCount; j++)
        {
           buffer.append( stringList[j] );
           buffer.append( separator );
        }

        buffer.append( stringList[stringCount] );

        return buffer.toString();
  }
```

R.STR.7 Avoid using `toUpperCase()` und `toLowerCase()` from `String`, because they create new objects. Before using excessive conversions, try to find out why they are needed. One case is comparing input against available data, possibly from a database. The solution could be to use a canonical form and to compare against it. Another possibility is to define simple helpers, which use the internal character array fro string processing.

R.STR.8 If a known set of string *constants* is frequently compared for equality, use their internal values. Calling `String.intern()` returns a string from a unique pool of string constants:

```
private static final String s = "…".intern();
if ( s == inputString ) { … }
```

R.STR.9 The use of the + operator to concatenate strings is very expensive, because intermediate string objects are generated.
Use `StringBuffer` and `append()` to concatenated pieces into a `String`. When the work is done, use `toString()` to convert the `StringBuffer` to an immutable string.

☞ Java 5 offers a class `StringBuilder` with the same methods of `StringBuffer`. This new class is *not* synchronized. All examples made for the `StringBuffer` can be applied to the new class, too.

When analyzing natural language sentences, it is better to resort to `java.text.BreakIterator` to break down a sentence into words.

R.STR.10 Cascade the calls to `append()` to show concatenation as a single operation (`buffer.append(…).append(…).append(…);).`

R.STR.11 If the `StringBuffer` is subsequently used (that is, after calling `toString()`), the original character array in that `StringBuffer` is copied into a new character array that is now referenced by the `StringBuffer`. The `String` object retains the reference to the *previously* shared character array.
Consequences:

- Concatenating elements using a local buffer and returning a string as result is fast and secure.
- To make the copying overhead occur at predictable times, execute `setLength()` on the buffer and set it to the predicted total length of the string to be altered. Applying this method makes the copying occur.

R.STR.12 Construct `StringBuffer` with an initial buffer size equal to the approximate maximum required.
If the elements are strings and known in advance set the buffer size to the total size of all strings[30]:

```
StringBuffer buffer =
   new StringBuffer( a.length() + b.length()
                     + … + z.length() );
```

[30] This solution uses the constructor `java.lang.StringBuffer( int length )`.

R.STR.13 Use the operations in `StringBuffer` to replace, delete and insert characters. The corresponding operations in `String` will cause the inefficient generation of many new objects.

R.STR.14 Don't reuse a `StringBuffer` for many string manipulations, that is, don't use `buffer.setLength(0)` but throw it away (eventualy setting `buffer = null`) and create a new one with the expected size.

R.STR.15 The `StringBuffer` class doesn't provide an `equals()` method. If such functionality is required, provide an helper method (declared as static, and part of an uninstantiable helper class). This method can be used in the `equals()` method of classes when comparing attributes of type `StringBuffer`.

```
public static isEqual( StringBuffer first,
                       StringBuffer second )
{
    if ( first == second )   return true;
    if ( first  == null ||
         second == null    ) return false;

    int length = first.length();
    if ( length != second.length() ) return false;

    for ( int i = 0; i < length; i++ )
    {
        if ( first.charAt(i) !=
             second.charAt(i)   ) return false;
    }

    return true;
}
```

4.1.4. Regular Expressions

Concept

A regular expression (RE)[31], often called a pattern, is an expression in string form that describes or matches a set of strings, according to certain syntax rules. Regular expressions are usually used to give a concise[32] description of a set, without having to list all elements. See [URL-7] for further information ressources.

There is no standard for the syntax of REs. Although many elements are common, different flavors exist.

Java implementation[33] - Matching

[31] Known abbreviations: RE, regexp, regex, or regxp, with plural forms regexps, regexes, or regexen.

[32] The origins of regular expressions lies in automata theory and formal language theory. See http://en.wikipedia.org/wiki/Regular_expression for a good and short introduction. The book mentioned in [URL-7] is an excellent traitment of the subject.

[33] A tutorial about the RegEx package is available from Sun: http://java.sun.com/docs/books/tutorial/extra/regex/index.html .

The Java version builds on the widely understood Perl 5 regular expression syntax[34], with some documented restrictions. The package consists of two classes (`Pattern` and `Matcher`) and an exception (`PatternSyntax-Exception`).

A `Pattern` is the object representing a compiled[35] regular expression.
A `Matcher` is the inspector used to determine the results of the match between the RE and the candidate string.
The matching process is depicted in the code fragment below:

```
try
{
    Pattern p = Pattern.compile( regularExpression );
    Matcher m = p.matcher( candidateString );
    …
}
catch( PatternSyntaxEception e ) { … }
```

Additionally, it is possible to specify a sub-expression that matches one of a range of possible patterns (*choice*). The alternative patterns are grouped by parenthesis and separated by a bar: `( RE1 | RE2 | … | REn ).`

Position markers are used to demand that certain characters occur in a certain position:

- `^` means "start of the line"
- `$` means "end of the line".

All special characters can be escaped, that is returned to their original meaning as in the alphabet, by preceding them with a backslash (e.g. `\$` means the character representing the dollar). The backslash itself is represented by `\\` .

Regular expressions are strings containing two kind of characters: *literal characters and meta characters.* "Literal" means "taken as it is", and "meta"[36] stands here for "hidden meaning of the character". Meta characters carry additional information about the characters to be matched in the candidate string.

The following character literals are defined:

Notation	**Character Literal**
`\C`	escape character `C`, where `C` is one of the characters used in REs: `\, ^, $, [, ], {, }, (, ), -, .`
`\0`*`N`*	character with octal value *N*; 0 <= N <= 7
`\0`*`NN`*	character with octal value *NN*; 0 <= N <= 7

[34] See http://www.perl.com/doc/manual/html/pod/perlre.html for the official documentation of the Perl 5 REs.

[35] "Compiled" means in this context the internal (with respect to the RE engine) representation of the regular expression. This form is more efficient in terms of processing than the visible string representation.

[36] "meta" comes from Ancient Greek and is a prefix (not a word on its own) which means "behind" or "hidden". *Meta* implies the part of something that is not immediately visible, is in the background, but which is there nonetheless and has an effect..

`\0`*`MNN`*	character with octal value *MNN*; 0 <= M <= 3, 0 <= N <= 7
`\x`*`HH`*	character with hexadecimal value HH
`\u`*`HHHH`*	character with hexadecimal value HHHH (Unicode)
`\t`	tabulator
`\n`	newline
`\r`	carriage return
`\f`	form feed
`\a`	alert (bell)
`\e`	escape
`\c`*`X`*	control character corresponding to *X* (`CTRL-X`)

The first kind of meta characters is the *character class*, which matches characters as elements of a set of characters that share some property:

The following character classes are defined:

Notation	**Character Class**
`. (dot)`	any character
`[X-Y]`	the set of characters between X and Y hint: if hyphen (`'-'`) and/or caret (`'^'`) are part of the search set, place the hyphen itself at the very beginning, and the caret at the very end of the RE
`[^X-Y]`	all characters excepted for the set `[X-Y]`[37] (see hint above)
`(RE)`	groups the subexpression, allowing to use quantifiers on it (see next table for the syntax of quantifiers) choice (alternation) is represented by `(RE1 | RE2 | … | REn)`
`\d`	one digit
`\D`	a non-digit
`\s`	whitespace[38] character
`\w`	a word character (letter, number, underscore)
`\W`	a non-word character

[37] In terms of set theory: the complement of the set described by [X-Y] with respect to the universe of discourse.

[38] The set of whitespace characters is composed by space, tab, newline, form feed, carriage return.

{lower}	lowercase alphabetic character (equivalent to [a-z])
{upper}	upper case alphabetic character (equivalent to [A-Z])
{alpha}	union of both sets, {lower} and {upper}
{alnum}	alphanumeric character (equivalent to [a-z0-9])
{punct}	punctuation character P = { !"#$%&'()*+,-./:;<=>?@[\]^_`{}~| }
{graph}	union of both sets, {alphanumeric} and {punctuation}
{print}	union of {graph} and <SPACE> character (*not* \s !)
{blank}	<SPACE> and <TAB> characters (*not* \s ! – so called "horizontal white space")
{xdigit}	hexadecimal digit (equivalent to [a-zA-Z0-9])

The other meta characters are *quantifiers*. They quantify the previous character's significance, by adding the concept of quantity to it. Quantifiers can be applied of subexpressions; a part of a complete RE is delimiter by parenthesis (e.g. *(RE)*) or by a character set ([x-y]) or by a character class ({alpha}).

The following quantifiers are defined:

Notation	**Quantifier**
?	minimum 0 occurrences, maximum 1 occurrence hint: this quantifier makes the previous character/group optional
+	minimum 1 occurrence, maximum unlimited hint: this quantifier makes the previous character/group required
*	minimum 0 occurrences, maximum unlimited hint: this quantifier makes the previous character/group optional
{n}	requires the previous character/group *exactly* n times
{n,}	requires the previous character/group *at least* n times
{n,m}	requires the previous character/group *at least* n times, *but no more than* m times

Java implementation - Searching

An alternative to the use of a Matcher is to call

- o split(String candidate) or
- o split(String candidate, int limit)

on the Pattern object. The split() method returns an array of strings found between successive matches of the compiled RE in the input string. The limit is the number of splits that will be sought.

Additionally, the `Matcher` provides Boolean methods which test the pattern on the candidate string, for exemple `lookingAt()` .

The `find()` method of `Matcher` looks for a substring that matches the RE, starting at the beginning of the input string and returning `true` if one is found. Sequential calls to `find()` will find each successive matching substring. After a successful match, the `start()` and `end()` methods can be used to find the index of the beginning and of the end of the substring. The `group()` method retrieves the actual matching substring.

Replacement operations (editing) on matched strings are also possible. The `String` class offers a bunch of methods built on the corresponding methods of `Pattern`.

R.REX.1 Starting from Java 1.4, use the `java.util.regex` package, not other solutions outside the JDK.

R.REX.2 Test[39] carefully the REs during development, trying to cover special and unusual cases.
Remember that the engine used in the `java.util.regex` package is by default "greedy", which means that the matcher returns the longest match possible. To do non-reedy matching, a question mark (`?`) must be added to the quantifier.

```
// greedy quantifiers
match = find("A.*c", "AbcAbc");         // AbcAbc
match = find("A.+", "AbcAbc");          // AbcAbc

// non-greedy quantifiers
match = find("A.*?c", "AbcAbc");        // Abc
match = find("A.+?", "AbcAbc");         // Abc
```

4.1.5. Mathematics

For a good overview of the current work in numerics for Java, consult http://math.nist.gov/javanumerics, afocal point for information on numerical computing in Java.

General issues

R.MAT.1 The `Math` class offers basic math functions, but no statistics. The companion class `StrictMath` has the same set of methods, and ensures that the operations will deliver the same results on all platforms. As with the modifier `strictfp` ("strict floating-point"), there is no increase in numerical precision, but only the guarantee that the results are not platform-dependent. "strictfp" means "strict according to the IEEE standard". The goal is to reach better performance, where strict reproducibility is not required. Please

[39] Because the Java implementation follows Perl's syntax (with some restrictions), it is possible to test the RE online at http://www.quanetic.com/regex.php .

note that only `java.util.BigDecimal` can deliver numerically correct results.

R.MAT.2 If complex mathematical operations are needed, consider the use of a library of mathematical functions.

R.MAT.3 Avoid `float` and `double` if exact results are required. Consider using `BigDecimal` or commercial mathematical libraries instead.

R.MAT.4 ***Never*** use `float` and `double` for monetary calculations.
Use a special `Money`[40] class or use `BigDecimal` from JDK.

R.MAT.5 ***Never*** compare real numbers (in Java: `float` or `double`) for equality. Use only "less than" and "greater than" for comparisons.

R.MAT.6 Cast operations on numerical data types (and `byte`) don't generate errors. Values outside the legal range are turned to other (wrong) values during casting.
This behavior can generate very subtle problems, especially in JDBC programming, because many drivers do not check for illegal results from cast operations.

Examples:
```
int  n = 100;
byte b = (byte) n;  // ok, range [-127,+128]
int  m = 200;
byte c = (byte) m;  // wrong, value is -56
```

One secure solution is to use `BigDecimal`:

```
double d = 0.3;
BigDecimal bd
   = new BigDecimal( Double.toString( d ) );
bd.setScale( scale, BigDecimal.ROUND_HALF_UP );

sqlStatement.setBigDecimal( columnIndex, bd );
```

In order to make the intention clear, a simple helper function can be realized:

```
public static long truncate( double number )
{
    return (long) number;
}
```

Because of type coercion, this function will also work on `float`. There is no need for overloading.

Utilities

R.MAT.7 In addition to standard arithmetic operations, BigInteger offers modulo arithmetic, functions for prime numbers, GCD and some other operations. Precise results are guaranteed for all methods.

R.MAT.8 The primary functionality for rounding is provided by the `round()` method in the Math package. This method rounds to the nearest whole number, delivering `long` as output (`2.0 to 2.0, 2.1`

[40] Java 1.4 and higher offers a `Currency` class representing all world currencies.

`to 2.0, 2.5 to 3.0`). In contrast, `rint()` delivers `double` as output. `0.5` will be rounded to the near even digit, resulting in 50% rounding up and 50% rounding down.

```
long   l = Math.round( d );
double d = Math.rint( d );
```

To round to *2* decimal places, use

```
d = Math.round( d * 100 ) ) / 100. ;
or
d = Math.rint( d * 100 ) / 100. ;.
```

The second method delivers a rounded result. For example, the results in the code above would be `1.35` for `round()` and `1.34` for `rint()`, if the input value `d` is `1.345` .

The method `ceil()` rounds to the next higher integer, `floor()` rounds to the next lower integer.

R.MAT.9 If rounding should be controlled by a given threshold, the following operation can be implemented. It returns the whole value rounded up or down, depending on the threshold.

```
public static long round( double number,
                          double threshold )
{
   long roundedValue = (long) Math.floor( number );

   if ((number - roundedValue) > threshold)
   {
      return (long) number + 1;
   }

   return (long) number;
}
```

Examples: with a threshold of `0.005`,
`347.010, 347.008, 347.006` ➔ `348` and
`347.005, 347.003, 347.000` ➔ `347`

R.MAT.10 For integer values, the *max, min, abs* functions can be implemented by simple on-liners:

```
int max = (a > b) ? a : b;
int min = (a > b) ? b : a;
int abs = (j > 0) ? j : -j;
```

R.MAT.11 The modulo operation (`a % b`, where `a MOD b` can be expressed as `(a - n * ( floor( a/n ))` delivers *mathematically* correct results only if `b > 0`. For negative value of a and b the results do not correspond to the laws of modular arithmetic.
An additional function `IEEEremainder()` takes a dividend and a divisor of type `double` and returns a `double`. The implementation follows the corresponding IEEE specification.

R.MAT.12 The "best" quality of randomness is delivered by `SecureRandom` in the package `java.security`. However, the mechanism is computationally expensive. When performance matters and the requirements on the random number generator are not too high, use `java.util.Random` instead.

R.MAT.13 If you need to generate random integers between `high` and `low`, use the following idiom:

```
Random r = new Random();
int j = (r.nextInt() & Integer.MAX_VALUE) %
        (high - low + 1) + low;
```

R.MAT.14 If you need to generate random integers between `0` and some upper bound, use the `nextInt( int j )` method in `java.util.Random` (available since version 1.2).
Use `j = n+1` to return values in the range `[0,n] (=[0,j[)`.

Additionally, use methods named `next<TYPE>()` to return random values for numbers other than integers (`long, float, double` – for real numbers, the result is in the range `[0.0,1.0[`).
To obtain random bytes, use:

```
  byte[] bytes = new byte[5];
  Random.nextBytes( bytes );
```

Use of BigDecimal

R.MAT.15 The `BigDecimal` class supports arithmetic with fixed-point[41] semantic, thus requiring a scale information for every arithmetic operation.
☞ In Java 5, the new `java.math.BigDecimal` class supports another semantic representation; floating point. This class has true floating-point operations consistent with those in the IEEE 754 revision.

R.MAT.16 In `BigDecimal`, `ROUND_HALF_EVEN` is the best general-purpose mode, especially when handling money amounts. It is what most people expect in real life during commercial exchanges.
This rounding mode, also called "round to nearest" is specified as the default rounding mode in IEEE 754.
5 is rounded up if the digit to the left of the five is odd. If the digit is even, five is rounded down.
This rounding mode miminizes cumulative error when applied repeatedly over a sequence of calculations.

R.MAT.17 Other rounding modes in `BigDecimal`:

- `ROUND_UP`: rounds away from zero
- `ROUND_DOWN`: rounds toward zero
- `ROUND_CEILING`: rounds towards positive infinity
- `ROUND_FLOOR`: rounds towards negative infinity
- `ROUND_HALF_UP`: 0..4 round down, 5..9 round up
- `ROUND_HALF_DOWN`: 0..5 round down, 6..9 round up

[41] In fixed-point computations, the number of fractional digits is a fixed constant.

`ROUND_UP` *always* increments the digit prior to a non-zero discarded fraction, `ROUND_DOWN` *never* increments the digit prior to a non-zero discarded fraction (truncation!).

If the number is positive, `ROUND_CEILING` behaves same as `ROUND_UP` and `ROUND_FLOOR` behaves same as `ROUND_DOWN`.

`ROUND_HALF_UP` and `ROUND_HALF_DOWN` compute the "nearest neighbor". The first one is the method generally taught in school.

Appendix 1

Basic information about floating-point arithmetic

Refer to [URL-7] for an introduction to this topic.

Binary floating-point number cannot represent common decimal fractions exactly. In a decimal system, the numerical values that can be represented are
`significand * 10`^{exponent},
while the numbers that can be represented in a binary system are
`significand * 2`^{exponent}.

In both cases, the significand and the exponent are integers. Therefore, whole integer values can be represented exactly in any integer base. In contrast, the set of fractional values that can be represented exactly by a finite number of digits (as it is the case on a computer), varies by base.

Examples

real number	**decimal system**	**binary system**	**binary fraction**
0.5	$5 * 10^{-1}$	$1 * 2^{-1}$	0.1_2
0.375	$375 * 10^{-3}$	$3 * 2^{-3}$	0.011_2
0.1	$1 * 10^{-1}$	N/A	$0.000110011\ldots_2$

The advantage of doing computation in decimal is that common used numbers can be represented exactly.

Numerical properties

property	**float**	**double**	**BigDecimal**
bits	24	53	---
decimal digits	≈ 6 to 9	≈ 15 to 17	1 to billions
exponent range	$[2^{-149}, 2^{127}] \approx [10^{-45}, 10^{38}]$	$[2^{-1074}, 2^{1023}] \approx [10^{-324}, 10^{308}]$	$[10^{-2147483647}, 10^{2147483648}]$

Appendix 2

Data Structures

The running time or time complexity of an operation on a collection is usually given in O notation ("big O"), as a function of the size `n` of the collection.

O notation	**Time behavior**	**Element operations** (accessing, adding, removing an element)
`O(1)`	constant time	very fast
`O(log n)`	logarithmic time (time proportional to the logarithm of `n`)	fast
`O(n)`	linear time (time proportional to `n`)	slow

For further information, please consult the literature reference at the beginning of the document (item [A-1]).

In the following table, `n` is the number of elements in the collection, `i` is an integer index, and `d` is the distance from an index `i` to the nearest end of a list, that is `min(i, n-i)`. The subscript `a` indicates amortized complexity: over a long sequence of operations, the average time per operation is `O(1)`, although any single operation could take time `O(n)`.

Operation	**LL**	**AL**	**HS, LHS**	**TS**	**HM,LHM**	**TM**
`add(o)`	O(1)	$O(1)_a$	$O(1)_a$	O(log n)		
`add(i,o)`	O(d)	$O(n-i)_a$				
`addFirst(o)`	O(1)					
`put(k,v)`					$O(1)_a$	O(log n)
`remove(o)`	O(n)	O(n)	O(1)	O(log n)	O(1)	O(log n)
`remove(i)`	O(d)	O(n-i)				
`removeFirst()`	O(1)					
`contains(o)`	O(n)	O(n)	O(1)	O(log n)		
`containsKey(o)`					O(1)	O(log n)
`containsValue(o)`					O(n)	O(n)
`indexOf(o)`	O(n)	O(n)				
`get(i)`	O(d)	O(1)				
`set(i,o)`	O(d)	O(1)				
`get(o)`					O(1)	O(log n)

Abbreviations: o – `java.lang.Object`, ji – `int` ji (index), k – key, v – value, LL – linked list, AL – array list, HS – hash set, LHS – linked hash set, TS – tree set, HM – hash map, LHM – linked hash map, TM – tree map.

This page was intentionally left blank.