		Código ARQ-EST-DB-001	
	Guía de Practica y Estándares de Desarrollo - Java EE	**Fecha de emisión 07.01.2015**	**Versión 01**

UNIQUE-YANBAL

Guía de Practica y Estándares de Desarrollo - Java EE

Estado : Versión Candidata
Confidencialidad : Público

YANBAL UNIQUE

Contenido

1. Historial de versiones

Autor/Responsable	Fecha	Versión/Comentarios
Jorge Cabrera	06/04/2015	Versión Candidata

2. Autores

Nombre	Rol
Martín Valdivia	Revisor/Colaborador - seguridad
Luis Alcántara	Revisor/Colaborador – CID
Joanna Mezaldi	Revisor/Colaborador – PAN
Luis Castro	Revisor/Colaborador – Área de Producción
Jorge Cabrera	Responsable/Elaborador

3. Lista de distribución

Nombre	Cargo /Rol
Centro de Integración y Desarrollo	
Área de Arquitectura y Nueva Tecnología	
Área de Infraestructura Soporte Técnico y Seguridad	

4. Área interesada/proyecto

El presente es elaborado por el área de Arquitectura y Nuevas Tecnologías (PAN), teniendo como áreas interesadas la organización IT corporativa así como todas las áreas corporativas (y de las empresas del grupo) interesadas en realizar en desarrollar componentes de base de datos (u aplicaciones que los incluyan).

5. Resumen ejecutivo

El cumplimiento de los presentes estándares y prácticas está relacionado directamente a las siguientes características en las aplicaciones:

1. Tiempo de desarrollo
2. Costo y tiempo de mantenimiento
3. Flexibilidad
4. Portabilidad
5. Re-uso
6. Reducción de las incidencias
7. Rendimiento
8. Costo
9. Soporte

Para su cumplimiento es imprescindible el compromiso de los líderes y responsables de la implementación, debiendo brindar los recursos y mecanismos necesarios para esto.

6. Introducción y objetivos

El objetivo del presente documento es definir los estándares y las prácticas **a ser cumplidas (de forma obligatoria)** en el desarrollo de componentes de Java y/ Java Enterprise Edition (JEE).

7. Definiciones

A continuación se enumeran las definiciones a ser tomadas en cuenta en el presente documento y anexos:

- ✓ Documento de Diseño Técnico: Documento que detalla el diseño técnico de un aplicativo o solución en particular.
- ✓ Documento de Análisis Técnico: Documento que detalla el análisis técnico de un aplicativo o solución en particular.
- ✓ Documento de Arquitectura: Documento que detalla la arquitectura de un aplicativo ó solución en particular.
- ✓ Práctica: Ejercicio realizado bajo ciertas reglas, para los fines de este manual se considera obligatoria.
- ✓ Estándar: Modelo, regla o patrón
- ✓ CID : Centro de integración y desarrollo corporativo Yanbal
- ✓ PAN: Área de proyectos , arquitectura y nuevas tecnologías corporativa de Yanbal
- ✓ ISS : Área de infraestructura, soporte y seguridad de sistemas corporativo
- ✓ CDI : Context Dependency Injection estándar que permite referenciar objetos definidos en el mismo contexto, parte de estándar JEE 6 y 7, considera los contextos de petición, aplicación, sesión, dependiente y conversación .

8. Documentos relacionados

El diseño de datos, así como de aplicaciones debe de enmarcarse dentro de lo especificado en los siguientes documentos, lo indicado en el presente no implica bajo ningún criterio el no cumplimiento de lo allí definido.

- ✓ GUIA DE SEGURIDAD DE SISTEMAS DE INFORMACION
- ✓ POLITICAS DE SISTEMAS DE INFORMACION
- ✓ POLÍTICAS GENERALES DE APLICACIONES
- ✓ PRINCIPIOS DE ARQUITECTURA
- ✓ POLITICAS DE ARQUITECTURA

9. Excepciones

Sin perjuicio de lo expuesto y reconociendo que una práctica y/o estándar no es aplicable a todos los proyectos y situaciones, **cualquier excepción al presente deberá de ser presentada, justificada y documentada mediante alguno de los siguientes mecanismos de acuerdo a su condición**:

1. Excepción aprobada por el proyecto y director del área de seguridad (ISS): Cuando la excepción vaya en perjuicio de los principios y políticas definidas por el área de seguridad.
2. Excepción aprobada por el proyecto y director del área PAN: Cuando la excepción vaya en perjuicio de los principios y políticas definidas por el área.

3. Detallada y justificada en el documento de arquitectura (o equivalente): Cuando se trate de una desviación a los presentes estándares.
4. Detallada y justificada dentro del documento de diseño técnico (o equivalente): En los puntos que específica hacerlo así el presente documento.

10. Estándares y prácticas

10.1. Principios de codificación:

Los principios de codificación a ser observados son:

a. KIS (keep it simple): El código debe de tratar de ser lo más simple posible, partiendo las tareas complejas en tareas sencillas y simples, evitando (en la medida de lo posible) código largo, sobre complicado y difícil de entender.

b. DRY: De las siglas " Don't Repeat Yourself", este principio indica que :

   a. Todo Comportamiento (o conocimiento "know how" [1]) debe tener una sola representación en código y debe de ser una sola pieza dentro del sistema.
   b. Las funciones comunes o lógica similar **no deben de ser duplicadas** (Ej: Copy + Paste).

   Por tanto, la lógica (comportamiento o "kwon how") debe ser encapsulada e incluida en componentes comunes de la aplicación (si son de su ámbito de interés) ó en la librería corporativa (ver punto i "Librería Corporativa") de tener probabilidades de re-uso entre aplicaciones.

c. YAGNY ("**Y**ou **A**ren't **G**onna **N**eed **I**t"): No se deben incluir características o funcionalidades que no sean necesarias para cumplir lo especificado - ***'si no está en el concepto, no debe de ser incluido'***.

d. Open/Closed principle: Este principio postula que las clases deben estar abiertas para ser extendidas, pero cerradas para su modificación.

e. Segregación de Interfaces[2]: Se debe considerar que: "Muchas interfaces específicas son mejores que una sola de uso genérico". Esto nos indica que ninguna clase cliente debería de depender de métodos que no utiliza, para esto es necesario dividir as tareas en interfaces más pequeñas, de esta manera no deberán "saber ó conocer" (depender) de métodos que no son de su interés.

f. Inversión de Dependencia : Se deben observar estos dos postulados :

   a. Los módulos de alto nivel no deben depender de los módulos de bajo nivel, ambos deben depender de abstracciones.

[1] Kwon How : Conocimiento sobre cómo se realiza una tarea o se consigue algo

[2] Interface : En este caso se refiere al estereotipo Interface, no a las interfaces entre sistemas

b. *Las abstracciones no deben depender de los detalles, los detalles deben depender de las abstracciones).*

El uso de estos principios debe de ser plasmado en los documentos de Arquitectura y Diseño, siendo el Área de Arquitectura la dirimente del cumplimiento de los mismos en caso de duda o conflicto y mediante la aprobación/elaboración de los documentos antes mencionados.

10.2. Consideraciones generales

Las siguientes consideraciones deben de ser tomadas en cuenta para toda aplicación o componente en lo que corresponda:

a. Todas las aplicaciones serán registradas en el catálogo de aplicaciones corporativo, esto será de responsabilidad del arquitecto líder asignado al proyecto o delegado.
b. Las aplicaciones serán de naturaleza Web, sin embargo los estándares presentados deben considerarse para aplicaciones utilitarias tipo cliente o consola dentro de la medida de lo aplicable, esto deberá de indicarse en el documento de arquitectura de la aplicación, el cual debe justificar dicha decisión.
c. Las aplicaciones deberán estar diseñadas para ejecutarse sobre servidor de aplicaciones IBM Websphere v 8.x.
d. El nivel de servlets a utilizar será 3.0.
e. Deberán empaquetarse en archivos ear o war, (no se deberá utilizar el tipo enhaced ear) y no deberán incluirse las fuentes del código.
f. Los archivos de librerías (jar, zip) deberán de ser externos a la aplicación y su ruta deberá de ser configurable como parámetro, además deberán de utilizase las ya existentes en el servidor de aplicaciones de existir y coincidir en la versión adecuada.
g. Las aplicaciones no deberán requerir la replicación de sesión para su correcto funcionamiento, salvo excepción formalizada en el documento de arquitectura.
h. Cada aplicación debe ser independiente en lo referente a configuraciones.
i. Las aplicaciones deberán seguir la siguiente estructura con respecto a sus configuraciones, librerías y logs :

```
/Apps/XXX/
        /Config
        /Lib
        /Logs
```

j. Los aplicativos deberán de ser “multipais”, es decir deberán brindar distintas configuraciones de mensajes, etiquetas y logos de acuerdo a la localización del usuario. Además de acuerdo a los requerimientos funcionales podrán existir funcionalidades que se comportarán distinto de acuerdo a este parámetro.
k. El nivel de los proyectos será JEE6
l. La JDK utilizada para desarrollo será 1.6
m. Los archivos de configuración y etiquetas estarán fuera del aplicativo (código fuente)

n. La ruta de los archivos de configuración deberá de ser configurable en un archivo de ubicación fija relativo a la aplicación, pero externa ella.
o. Los archivos de configuración seguirán el siguiente patrón para sus parámetros:
    a. Valor=parámetro
p. No deben existir parámetros en el código fuente ‘hardcode’, dichos valores deben estar definidos en archivos de configuración ó alternativamente en la base de datos en caso estos tengan altas probabilidades de cambiar y se requiera hacerlo mediante funcionalidad o sin detener el aplicativo, en cuyo caso la aplicación deberá de refrescar dicha configuración de manera inmediata.
q. Las interfaces con otros sistemas deberán de ser parametrizables en cuanto a sus datos (url, puerto, nombre de archivo, etc) en lo que a cada una corresponda de acuerdo a su definición; esto se realizará mediante los archivos de configuración y deberá de estar formalizado en el Documento de Diseño.
r. Todos los archivos fuente y de configuración deben tener como encoding UTF-8
s. Se utilizará el log (archivo de auditoria) para cualquier tipo de mensaje que la aplicación desee imprimir para este fin, no debiendo existir llamadas directas a System.out.println o Systemerror (en caso requerir mensajes para debug utilizar la clase Log en nivel ‘Debug’ o las herramientas de Depuración de código (‘Debuggers’)
t. No deben existir ‘imports’ innecesarios dentro de los archivos de código fuente.
u. Se deberán crear scripts de comandos (Shell scripts) para la creación de directorios, archivos y configuraciones de S.O de ser el caso.
v. Las aplicaciones utilizarán conexiones tipo pool registradas como ‘Datasources’ en el servidor de aplicaciones.
w. El máximo de complejidad permitido en un método será menor a 13, validándose en su complejidad ciclomática (mediante la herramienta SONAR), salvo excepción de diseño formalizada en el documento de diseño.
x. No deberán de existir porciones de código repetidas, esto se validará mediante la herramienta SONAR, característica “Duplicated Blocks”.
y. El manejo de Threads ó hilos (de ser necesario) para tareas donde el procesamiento de este tipo sea conveniente se implementará mediante clases directamente (sin adoptar una librería para este fin), esto deberá estar indicado en el Documento de Arquitectura y descrito en detalle en el Diseño Técnico.
z. Las aplicaciones utilizarán los archivos pages.xml para definir la navegación, así como las clases “Controller” para especificar el ‘outcome’ al cual deben dirigirse.

10.3. Librería corporativa

Las funciones y entidades que sean (o tengan altas probabilidades de ser) utilizadas en más de una aplicación deberán de añadirse a esta librería, siguiendo este mismo criterio los frameworks (y/ó librerías) externas que se decidan incorporar ya sean libres o de fabricantes deberán estar encapsuladas dentro de esta librería.

Esta librería debe de ser incluida en los documentos de Arquitectura y Diseño indicando la versión de la misma además debe especificar que métodos y clases deberán de ser modificadas y/o añadidas a la misma (en la etapa diseño).

1. **Criterio de Adopción:** La decisión de adopción o no pertenecerá al área de arquitectura detallándose la incorporación en el documento de diseño y en la documentación correspondiente de la Librería Corporativa.

2. **Criterios de uso:** Las aplicaciones podrán utilizar esta librería utilizando los siguientes métodos:
   - Mediante las Interfaces
   - Extendiendo sus clases

No se podrán hacer llamadas directas y/o instancias objetos concretos de la misma, como por ejemplo:

Incorrecto:

```
public interface A
{
}
public class B implements A
{
}

public static void main(String[] args)
{
    B test = new B();
 }
```

Correcto:

```
public interface A
{
}
public class B implements A
{
}

public static void main(String[] args)
{
    A test = new B();
    //A test = new A(); // no compilará
}
```

- **Esta librería podrá ser utilizada por proveedores y proyectos internos, ofreciéndose "As-Is" (como esta), siendo el proveedor ó proyecto responsable de las nuevas funciones y/o correcciones necesarias para su correcto funcionamiento.**

10.4. Estilo de programación

Como estilo de programación se decide adoptar el estilo de programación de Google, el cual se refiere en el Anexo C –Java Google Style, sin prejuicio a esto:

1. Los nombres de los métodos estarán escritos en español, salvo donde sea conveniente la adopción de un estándar de nombre en inglés debido al uso de "Reflection" u otra librería o capacidad JEE que así lo requieran, como por ejemplo los nombres getXXX o setXXX.

2. El número máximo de parámetros será de cinco (5), de sobrepasar este número se deberá de utilizar un objeto complejo tipo POJO.

3. La nomenclatura de paquetes Java será:

   Para librerías corporativas:

   - com.yanbal.corp.xxx

   Para las aplicaciones:

   - com.yanbal.corp.<aplicacion>.xxxx.xxx

   Debiendo tener al menos 3 'grados' de profundidad, los paquetes deben estar descritos a detalle en el documento de diseño y arquitectura según corresponda.

10.5. Uso de patrones

Los patrones a ser identificados e implementados en el desarrollo del código fuente se encuentran en el Anexo A – Relación de Patrones, el uso detallado de dichos patrones serán definidos durante el diseño del aplicativo al detallar el comportamiento de sus componentes.

De la misma manera en el Anexo B – Relación de Anti Patrones, se encuentran las malas prácticas que deben ser evitadas durante las etapas de diseño y desarrollo.

10.6. Estructura de paquetes

Las aplicaciones deberán presentar una división clara entre lógica de negocio, presentación (interface de usuario) y persistencia, la estructura de paquetes que deben seguir es:

1. **com.yanbal.<aplicacion>.aplicacion :** Aquí se encontrarán las clases que manejarán la aplicación y su comportamiento en sí misma, acorde a esto estarán encargadas de inicializar su entorno y valores principales de la misma, como por ejemplo :

   - ✓ Parámetros de la clase Log
   - ✓ Parámetros de las bases de datos del sistema
   - ✓ Parámetros y bases de datos de las interfaces
   - ✓ Parámetros de funcionamiento

   En esta capa se debe observar:

   - ✓ Debe existir una única instancia de la clase Aplicación, la cual inicializara los objetos principales de la aplicación tales como Log, Parámetros, Base de Datos (Ej: utilizando la etiqueta @Startup).
   - ✓ Las clases llevarán la etiqueta @Stateless
   - ✓ La clase aplicación debe de implementar el patrón singleton (Ej : @Singleton)
   - ✓ Los objetos que inicialice serán utilizados por las otras capas mediante el patrón ''Factory' o CDI (inyección de contexto).

2. **com.yanbal.<aplicacion>.entities :** Contendrá los objetos que representan el modelo de datos relativos a la aplicación, así como sus propiedades y los métodos necesarios para acceder a ellas. Estos serán utilizados como 'moneda' entre las capas del aplicativo y contendrán los datos sobre la persistencia y mapeo a la base de datos, se debe observar :

    - ✓ Llevaran el nombre de la entidad que representan ej : Persona
    - ✓ Deben utilizar la etiqueta @Name
    - ✓ No deberán definir un contexto ("scope")
    - ✓ No deberán estar en el contexto de la aplicación o sesión del usuario directamente.
    - ✓ No deben de utilizarse directamente para exponer o consumir interfaces como web services, jms ó rmi (si pueden extenderse para este fin).
    - ✓ Deben implementar un constructor default (vacío) y otro con todas sus propiedades
    - ✓ Deben implementar los métodos equals () y toString()
    - ✓ Solo deben contener etiquetas y lógica relacionada a la persistencia de los datos que contienen
    - ✓ Deberán de implementar las restricciones relacionadas a la validación de campos (javax.validation.constraint) en relación a la base de datos.
    - ✓ No deberán de ser serializables.

3. **com.yanbal.<aplicacion>.vista.beans :** Contendrán los POJO (objetos planos con propiedades y los métodos para acceder a ellas : setters y getters) , específicos a una vista o varias y se utilizarán cuando sea necesario o practico a comparación de utilizar una entidad directamente (ver 2), podrán contener objetos entities cuando sea conveniente.

4. **com.yanbal.<aplicacion>.controllers:** Estas clases estarán encargadas de manejar las llamadas de las interfaces de usuario y ejecutar la operación de negocio correspondiente, asi como definir los 'outcomes' relacionados a la navegación.

    - ✓ Llevarán la etiqueta @Stateless
    - ✓ No implementarán lógica de negocio
    - ✓ Llevarán un nombre relativo a la funcionalidades a las cuales sirven, con el postfijo Controler , El GestionParametrosControler
    - ✓ Podrán contener los campos y objetos representación de la interface de usuario
    - ✓ Indicarán un 'outcome' o salida cuando sea necesario, esta será manejada por otro componente encargado de la navegación.
    - ✓ Utilizarán inyección (CDI) para acceder los objetos de los contextos de: aplicación, conversación, sesión y request.

5. **com.yanbal.<aplicacion>.bo:** Aquí encontraremos las clases que implementarán las operaciones 'de negocio' es decir las funciones que las clases de control de la aplicación llamaran a fin de completar la funcionalidad requerida. En este paquete debemos observar :

- ✓ Deberán de ser Stateless
- ✓ Deberán de terminar en el postfijo BO , ej : PersonaBO
- ✓ La operación de negocio debe de ser completa y atómica , debiendo contener llamadas a métodos en caso realice operaciones complejas y/o re-utilizables
- ✓ Deberán ser accedidas utilizando el patrón Factory
- ✓ Expondrán métodos públicos para ser llamados desde las interfaces de usuario
- ✓ No deberán tener campos u objetos que sean representación de la interface de usuario

6. **com.yanbal.<aplicacion>.dao** : Este paquete deberá implementar los métodos de lectura y persistencia de los objetos beans:

   - ✓ Utilizarán la librería MyBatis para la persistencia
   - ✓ Serán las únicas con lógica de persistencia, lectura de datos y dependencia con las librerías de persistencia.

   En el caso del uso de JDBC para la ejecución de comandos no encapsulados en Procedimientos Almacenados (SP), deberá de considerarse una excepción formalizada en el documento de diseño, además se utilizarán clases implementadas en la librería corporativa para realizar las llamadas a dichos componentes, esto deberá de ser indicado como una excepción en el documento de diseño y/o arquitectura de ser caso.

7. **com.yanbal.<aplicacion>.session:** En este paquete se encontrarán los objetos que se almacenaran en la sesión del usuario, debemos considerar :

   - ✓ Podrán extender de las clases “entitites”.
   - ✓ Utilizaran etiquetas para el contexto y nombre (@Name , @SessionScoped)
   - ✓ Se accederán mediante CDI
   - ✓ No deberán de superar un tamaño de 10 Kilobytes por usuario en ningún momento de ejecución de la aplicación.

8. **com.yanbal.<aplicacion>.exception :** En este paquete se deberán de crear las clases para el manejo de excepciones de negocio.

9. **com.yanbal.<aplicacion>.interfaces.<tipo>.servicios :** En este paquete se deberán de crear las clases encargadas del manejo de interfaces con otros sistemas.

10. **com.yanbal.<aplicacion>. interfaces. <tipo>.beans:** En este paquete se deberán crear las clases de manejo de datos para las interfaces con otros sistemas.

10.7. Manejo de Excepciones:

Las aplicaciones deben controlar adecuadamente los errores y excepciones que puedan ocurrir (excepciones esperadas) mostrando el mensaje adecuado y utilizando el ‘log’ (archivo de auditoria), por tanto el código debe asegurar no ‘ocultar las excepciones o errores’.

A continuación se detallan las capas y puntos donde deben manejarse las excepciones:

1. **Navegación:** El archivo de navegación deberá de capturar las excepciones (de cualquier tipo), mostrando una página de error genérica, la página de error deberá de dejar el detalle de la excepción en el log de eventos.

2. **Aplicación:** La capa de aplicación deberá manejar todos los errores que puedan ocurrir (Catch Exception), mostrando una pantalla (ó página) de error genérica al aplicativo, dicha pantalla deberá de dejar una entrada en el log de auditoria correspondiente con el detalle de la excepción atrapada.

3. **Capa de Control :** Esta capa deberá manejar todos los errores que puedan ocurrir(Catch Exception) , mostrando una pantalla (o página) de error genérica, dicha pantalla deberá de dejar una entrada en el log de auditoria correspondiente con el detalle de la excepción atrapada

4. **Capa de Negocio:** Todos los métodos de negocio deben manejar las excepciones '**esperadas'** en la operación que realizan y, mostrar el mensaje adecuado en la pantalla y dejar una entrada en el log de auditoria, a continuación se presenta una lista (no restrictiva), de las excepciones que esta capa debe manejar.

   a. Errores de validación (ingreso y características de los datos ingresados)
   b. Errores por validaciones o lógica de negocio
   c. Errores de Base de Datos (Ej : JDBC, JPA)
   d. Errores de Lectura de Archivo
   e. Errores sobre Datos no encontrados y/o inválidos (Ej : objetos nulos)

   Esta capa deberá realizar las validaciones de negocio (reglas y accesos) y de entidades ('bean validation'), así como de ingreso de datos, pudiendo propagar una excepción a la capa siguiente o mostrando el mensaje adecuado en la página correspondiente. Para las excepciones **no esperadas,** deberá delegar el manejo a la capa superior (aplicación) propagándola a la capa superior (rethrow).

5. **Llamadas a componentes externos al entorno de ejecución:** Como pueden ser :

   1. La ejecución de sentencias HSQL /JPQL/PROCEDIMIENTOS ALMACENADOS Y/O SENTENCENCIAS SQL
   2. Llamadas a : Web Services/RMI/CORBA/EJB
   3. Llamadas al sistema operativo (o scripts que se ejecuten en otro contexto como bat, sh)

   Se deben de controlar todas las excepciones (Catch Exception), realizando lo siguiente:
      1. Dejar una entrada en el 'log', con la excepción original detallada.
      2. Propagar la excepción a la siguiente capa (re throw)

10.8. Reportes

Los reportes operativos de las aplicaciones (no correspondientes al área de inteligencia de negocio o BI) se implementarán utilizando:

1. Procedimientos Almacenados para obtener los datos
2. Jasper Reports (la versión será especificada en el documento de arquitectura)

10.9. Motor de Reglas

Las aplicaciones deberán de considerar utilizar el motor de reglas (Drools.jar), en los siguientes casos:

- ✓ Lógica con razonables probabilidades de cambiar (así no sea frecuentemente)
- ✓ Lógica compleja
- ✓ Lógica que varía de acuerdo al usuario y/ o ubicación geográfica (país, localización o geografía)
- ✓ Evitar el uso y proliferación de parámetros de comportamiento en la configuración de la aplicación
- ✓ No debe utilizarse para el manejo de datos 'sincronizados' o que presenten un estado, estado como por ejemplo inventarios u stock.
- ✓ Las reglas no deben ejecutar métodos de negocio directamente, se debe utilizar el patrón 'Command', indicando una lista de acciones a realizar por la aplicación.

Como por ejemplo:

- ✓ Oferta Comercial
- ✓ Reglas de Negocio
- ✓ Validaciones Complejas
- ✓ Comportamientos por país

Los métodos para el uso de este componente se encuentran en las librerías corporativas, sin embargo se debe considerar:

- ✓ Las llamadas deben de estar encapsuladas en clases particulares de gestión de reglas
- ✓ La ubicación de (los) paquetes deberá de ser un parámetro de la aplicación
- ✓ Se deben utilizar beans como hechos (facts) u objetos POJO 'adoc' para el caso
- ✓ No deben existir llamadas fuera de estas clases
- ✓ Debe de mantenerse el enfoque de 'caja negra' durante su implementación

10.10. Interfaces de usuario

Las aplicaciones interfaces con el usuario deberán obedecer los siguientes puntos, salvo en los casos donde los navegadores definidos no soporten dicha práctica u estándar.

1. Se favorecerá el diseño de página única (Single Page)
2. Las páginas fuentes de los aplicativos deberán de ser xhtml
3. Solo se deberán utilizar librerías (tags) estándar de la librería JSF 2.2 (no utilizar tags propios de la librería jsf o implementación escogida)
4. Se favocera el uso de AngularJS para :
   a. Llamadas AJAX, JSON o REST
   b. Transiciones visuales

c. Ocultar/Mostrar elementos del DOM
d. Generar HTML dinámicamente
e. Ordenamiento de Tablas
f. Búsquedas
g. Otros casos que la librería implemente

5. Se podrá utilizar jslt en los casos donde los componentes JSF ó AngularJS no sean adecuados.
6. No se deberán utilizar librerías (tags) propietarias o pertenecientes a librerías libres.
7. Las páginas deberán de seguir el estándar HTML 5, mostrando una clara diferenciación de contenido y estilo.
8. El estándar HTML 5 no será obedecido en los casos donde los navegadores definidos para la certificación de la aplicación no los soporten adecuadamente, debiendo reconocerse el navegador cliente y generando el código adecuado para el mismo.
9. El encoding de las páginas resultante (enviada al navegador) deberá de ser UTF-8
10. Deberán de hacer uso extensivo de AJAX y la carga de datos, salvo en los casos donde por performance y usabilidad resulte conveniente.
11. Las paginas deberán de generar un archivo HTML esqueleto (sin atributos) (ver Anexo 01 – HTML Y CSS)
12. Deberá de utilizar una hoja de estilos para definir los atributos de los elementos (ver Anexo 01 – HTML Y CSS)
13. Los textos deberán de estar externalizados en archivos de texto y deberán estar dentro de los archivos de configuración de la aplicación y no empaquetada junto a su código fuente.
14. Las páginas deberán de soportar 'internalización' cargando las etiquetas de acuerdo al idioma definido en el navegador, o uno default de no poder obtenerse o no estar definidas dichas etiquetas.
15. Todas las operaciones presentarán un mensaje de confirmación antes de ser ejecutadas.

10.11. Interfaces

Las interfaces con otros sistemas, soluciones u aplicaciones podrán ser de los siguientes tipos, sin embargo su especificación será aprobada por arquitectura en el documento de diseño:

a. Web Services (implementados en los protocolos SOAP y REST).
b. Message Queue : Mediante el estándar JEE 7

Para el caso específico de las interfaces se deberá imprimir en el log de auditoría:

1. Referencia: ID de transacción
2. Mensaje : Código respuesta - Mensaje de respuesta

En el caso de ser necesarias cargas o transferencias masivas se utilizará InfoSphere Datastage como herramienta ETL, según los estándares definidos para los mismos.

10.12. Transaccionalidad

El código y manejo de excepciones de las aplicaciones deberá:

1. Asegurar no dejar datos inválidos, salvo falla catastrófica que no pueda ser controlada en el contexto de la aplicación.
2. Para las operaciones atómicas de base de datos, estas deberán de estar encapsuladas en un procedimiento almacenado (Store Procedures).
3. La transaccionalidad para las interfaces será implementada de la siguiente manera:

    a. Con la(s) operación(es) compensatoria en los casos donde la transacción este fuera del contexto del contenedor Ej. : Archivos Planos , Servicios Web ; para este fin considere :
        a. No deberán de existir más de 2 sistemas involucrados
        b. Asegurar no dejar datos inválidos
        c. Realice la operación en el contexto donde se encuentre primero, sin confirmar (UNCOMMITED), de ser necesario esto se realizará programáticamente.
        d. Realice la operación fuera del contexto donde se encuentra al final
        e. Atrape todos los tipos de excepciones en la llamada y realice el re-intento
        f. Defina un mecanismo de alerta al usuario y al administradores del sistema (y/o interesados) para alertar de manera proactiva sobre fallos en estas interfaces.
        g. Se debe asegurar no dejar datos inválidos en ambas entidades, salvo falla catastrófica que no pueda ser controlada en el contexto de la aplicación.

4. En el caso de transacciones que incluyan operaciones fuera del contenedor (ej: llamadas a web services, colas), se deberá considerar dentro del listado de caso de pruebas una o varias pruebas forzando el fallo, para comprobar que el mecanismo funciona adecuadamente.

10.13. Auditoria

Las aplicaciones deberán implementar las siguientes normas de auditoria:

1. Se utilizarán las clases de Log, en el nivel y configuración adecuada
2. El archivo de auditoría será único por aplicación y deberá de ser configurable mediante parámetros ya sea mediante archivo plano o tabla de base de datos.
3. El nombre del archivo de auditoría seguirá el siguiente formato XXXXX.YYYY-MM-DD.log (<NombreApp><YYYY.MM.DD>.log) como por ejemplo : ARIALWEB.2011-05-13.log
4. En el caso de los archivos de auditoría se deberán considerar los siguientes parámetros respectivos al archivos:
    ✓ Tamaño Máximo : Tamaño máximo antes de cerrar el archivo
5. Las clases de auditoría estarán implementadas dentro de la librería corporativa.
6. Las operaciones (ver numeral 11 de la presente lista), excepciones y errores del sistema deben guardarse en el archivo de auditoria (log), el cual contendrá los siguientes campos (que sean aplicables) en el siguiente orden :
    a. Timestamp (fecha, hora, min, seg) de la entrada (:MM:SS sss)
    b. Operación : Operación que está realizando el sistema

c. Aplicación : Nombre de la aplicación
d. Usuario : Usuario que realiza la operación
e. Clase : Nombre de la clase u objeto (class)
f. Dirección ip (de la sesión) : dirección ip de la sesión http del usuario
g. Mensaje : Cuerpo del mensaje
h. Tipo : Clasificación del mensaje (Debug, Error, Fatal, Info, Warn)
i. Referencia al objeto principal (Ej identificador)

7. Las aplicaciones especificarán un parámetro para elegir el nivel de del archivo de auditoria (al iniciar la aplicación), pudiendo elegir que mensajes se ingresarán en el mismo por niveles.
8. Durante la codificación se considerarán las llamadas necesarias para todos los niveles de auditoría de acuerdo a los requerimientos del caso.
9. De la misma manera los siguientes eventos deben de ingresar en el archivo de auditoria: ingreso, salida, errores, intentos de ingreso no exitosos.
10. No se deberán utilizar las sentencias System.out ó System.error
11. Las aplicaciones deberán dejar mensajes en el archivo de auditoría en los siguientes tipos de operaciones :
    - ✓ Modificación/Actualización /Ingreso de datos
    - ✓ Lectura de datos (solo en el caso de datos personales ó sensibles de negocio)

11.1. Seguridad

El código fuente de asegurar cumplir:

a. Las aplicaciones deberán de cumplir lo establecido en la **GUIA DE SEGURIDAD DE SISTEMAS DE INFORMACION.**
b. Las aplicaciones utilizarán el estándar JAAS para autorización
c. La seguridad se basará en roles y será declarativa, mediante el uso de etiquetas ó tags, como por ejemplo :

```
@ServletSecurity(
@HttpConstraint(transportGuarantee =
TransportGuarantee.CONFIDENTIAL,
    rolesAllowed = {"TutorialUser"}))
```

d. La aplicación deberá tener un rol de administrador técnico, mismo que tendrá acceso a :
    a. Visualizar el archivo (ó tabla) de auditoria (se dederá considerar una funcionalidad para este fin)
    b. Cambiar los parámetros del sistema (de estar especificado funcionalmente);
    c. Desbloquear cuentas, salvo aquellas que utilicen el portal (o un directorio activo), para autenticar a los usuarios.
    d. Asignar contraseñas temporales (expiradas) a los usuarios, salvo aquellas que utilicen el portal (o un directorio activo), para autenticar a los usuarios.
e. Los datos de conexión a la base de datos y otro sistemas deberán de estar encriptados y almacenados fuera de la aplicación.

f. La autenticación se realizará contra el LDAP para las aplicaciones internas, es decir para los usuarios internos a la organización.
g. Para las aplicaciones externas (usuarios no inscritos en el LDAP corporativo) se utilizará una base de datos en donde las contraseñas deberán estar encriptadas.
h. El estándar de encriptación se indica en la de acuerdo a la **GUIA DE SEGURIDAD DE SISTEMAS DE INFORMACION** y los parámetros de encriptación serán considerados como parámetros de la aplicación en sus archivos de configuración.
i. La autorización se dará a nivel de funcionalidad (un rol tendrá acceso a un grupo determinado de funcionalidades), las aplicaciones deberán de especificar la forma en que se gestionarán dichos accesos.
j. Los roles solo tendrán acceso a los datos y funcionalidades requeridas para realizar sus funciones.
k. Se deberá validar el rol antes de mostrar las interfaces correspondientes a cada funcionalidad (autorización por página y opción de menú).
l. Las validaciones de datos se deben realizar en el servidor de aplicaciones, estas se realizarán mediante la etiquetas en los Beans (Bean Validation) y en la capa de negocio o aplicación, sin embargo se podrán realizar validaciones *también* en la capa de presentación cuando sean convenientes para la estabilidad y experiencia del usuario.
m. Las validaciones deberán de ser completas (tamaño, tipo de dato, longitud) y deben actuar sobre todos los campos
n. No se debe permitir el ingreso de los caracteres : < > " ' % ( ) & + \\' \"
o. Los campos no permitirán el ingreso de caracteres distintos a los valores validos (Ej: un campo tipo numérico no permitirá en ingreso de caracteres salvo números, para esto utilizarán los eventos de ingreso de teclado (Ej : onkeypress, onkeyup).
p. Las aplicaciones solo utilizarán bases de datos internas a la corporación
q. Las aplicaciones externas estarán aseguradas los protocolos indicados en la **GUIA DE SEGURIDAD DE SISTEMAS DE INFORMACION.**
r. Las aplicaciones solo permitirán la subida de archivos provenientes de usuarios autenticados
s. Las interfaces expuestas y consumidas por la aplicación serán solamente externas
t. Los archivos que las aplicaciones podrán aceptar son :
    a. xls, pptx,doc, docx, txt, jpg, bmp. tiff , ppt, pdf, mp3, mp4, avi.
u. Para los archivos fuentes de datos (xls, txt) se procesarán en memoria antes de almacenarse y solo se almacenaran de ser requerido)
v. En las aplicaciones internas las contraseñas deberán bloquearse luego de un número parametrizable de intentos no exitosos
w. Las pantallas de error solo mostrarán el mensaje de error, no mostrando información adicional sobre la excepción y su pila, dicha información se guardará en el log.

12. Herramientas

Las herramientas que se pueden utilizar para llevar a cabo las prácticas y asegurarlas son: (la lista presentada no es limitativa, obligatoria ni restrictiva y su uso no debe de afectar los estándares definidos en el presente, de la misma manera el proyecto definirá las versiones a utilizar de ser necesario:

1. Eclipse IDE (y sus plugins)
2. Google CodePro AnalityiX
3. https://code.google.com/p/google-styleguide/source/browse/trunk/eclipse-java-google-style.xml
4. Hibernate Tools
5. Db2 Explain Tools
6. SonarQube
7. Aqua Studio
8. IBM DB2 Control Center (Centro de Control)
9. Jasper Reports Designer

13. Fuentes

1. http://www.artima.com/intv/dry.html
2. http://www.objectmentor.com/resources/articles/dip.pdf
3. http://www.hongkiat.com/blog/building-html5-css-webpages/
4. https://developers.google.com/java-dev-tools/codepro/doc/requirements
5. http://docs.oracle.com/javaee/6/tutorial/doc/gjbbk.html

Anexo 01 - HTML Y CSS

```
<!doctype html>
<head>
  <meta charset="utf-8">
  <title>Our First HTML5 Page</title>
  <meta name="description" content="Welcome to my basic template.">
  <link rel="stylesheet" href="css/style.css?v=1">
</head>

<body>
    <div id="wrapper">
        <header>
            <h1>Welcome, one and all!</h1>

            <nav>
                <ul>
                    <li><a rel="external" href="#">Home</a></li>
                    <li><a rel="external" href="#">About us</a></li>
                    <li><a rel="external" href="#">Contacts</a></li>
                </ul>
            </nav>
        </header>

        <div id="core" class="clearfix">
            <section id="left">
                <p>some content here.</p>
            </section>

            <section id="right">
                <p>but some here as well!</p>
            </section>
        </div>

        <footer>
            <p>Some copyright and legal notices here. Maybe use the © symbol a bit.</p>
        </footer>
    </div>

</body>
</html>
```

```
!doctype html>
<head>
  <meta charset="utf-8">
  <title>Our First HTML5 Page</title>
  <meta name="description" content="Welcome to my basic template.">
  <link rel="stylesheet" href="css/style.css?v=1">
</head>

<body>
    <div id="wrapper">
        <header>
            <h1>Welcome, one and all!</h1>

            <nav>
                <ul>
                    <li><a rel="external" href="#">Home</a></li>
                    <li><a rel="external" href="#">About us</a></li>
                    <li><a rel="external" href="#">Contacts</a></li>
                </ul>
            </nav>
        </header>

        <div id="core" class="clearfix">
            <section id="left">
                <p>some content here.</p>
            </section>

            <section id="right">
                <p>but some here as well!</p>
            </section>
        </div>

        <footer>
            <p>Some copyright and legal notices here. Maybe use the © symbol a bit.</p>
        </footer>
    </div>

</body>
</html>
```

Extraído de http://www.hongkiat.com/blog/building-html5-css-webpages/

ANEXO A – Relación de Patrones

Extraído de: http://www.blackwasp.co.uk/GofPatterns.aspx

Gang of Four Design Patterns

by Richard Carr, published at http://www.blackwasp.co.uk/GofPatterns.aspx

The Gang of Four are the four authors of the book, "Design Patterns: Elements of Reusable Object-Oriented Software". In this article their twenty-three design patterns are described with links to UML diagrams, source code and real-world examples for each.

What are Design Patterns?

Design patterns provide solutions to common software design problems. In the case of object-oriented programming, design patterns are generally aimed at solving the problems of object generation and interaction, rather than the larger scale problems of overall software architecture. They give generalised solutions in the form of templates that may be applied to real-world problems.

Design patterns are a powerful tool for software developers. However, they should not be seen as prescriptive specifications for software. It is more important to understand the concepts that design patterns describe, rather than memorising their exact classes, methods and properties. It is also important to apply patterns appropriately. Using the incorrect pattern for a situation or applying a design pattern to a trivial solution can overcomplicate your code and lead to maintainability issues.

Who are the Gang of Four?

The *Gang of Four* are the authors of the book, "Design Patterns: Elements of Reusable Object-Oriented Software". This important book describes various development techniques and pitfalls in addition to providing twenty-three object-oriented programming design patterns. The four authors were Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

Gang of Four Design Patterns

This section gives a high-level description of the twenty-three design patterns described by the Gang of Four. Each pattern description includes a link to a more detailed article describing the design pattern and including a UML diagram, template source code and a real-world example programmed using C#.

Creational Patterns

The first type of design pattern is the *creational* pattern. Creational patterns provide ways to instantiate single objects or groups of related objects. There are five such patterns:

- **Abstract Factory**. The abstract factory pattern is used to provide a client with a set of related or dependant objects. The "family" of objects created by the factory are determined at run-time.
- **Builder**. The builder pattern is used to create complex objects with constituent parts that must be created in the same order or using a specific algorithm. An external class controls the construction algorithm.

- **Factory Method**. The factory pattern is used to replace class constructors, abstracting the process of object generation so that the type of the object instantiated can be determined at run-time.
- **Prototype**. The prototype pattern is used to instantiate a new object by copying all of the properties of an existing object, creating an independent clone. This practise is particularly useful when the construction of a new object is inefficient.
- **Singleton**. The singleton pattern ensures that only one object of a particular class is ever created. All further references to objects of the singleton class refer to the same underlying instance.

Structural Patterns

The second type of design pattern is the *structural* pattern. Structural patterns provide a manner to define relationships between classes or objects.

- **Adapter**. The adapter pattern is used to provide a link between two otherwise incompatible types by wrapping the "adaptee" with a class that supports the interface required by the client.
- **Bridge**. The bridge pattern is used to separate the abstract elements of a class from the implementation details, providing the means to replace the implementation details without modifying the abstraction.
- **Composite**. The composite pattern is used to create hierarchical, recursive tree structures of related objects where any element of the structure may be accessed and utilised in a standard manner.
- **Decorator**. The decorator pattern is used to extend or alter the functionality of objects at run-time by wrapping them in an object of a decorator class. This provides a flexible alternative to using inheritance to modify behaviour.
- **Facade**. The facade pattern is used to define a simplified interface to a more complex subsystem.
- **Flyweight**. The flyweight pattern is used to reduce the memory and resource usage for complex models containing many hundreds, thousands or hundreds of thousands of similar objects.
- **Proxy**. The proxy pattern is used to provide a surrogate or placeholder object, which references an underlying object. The proxy provides the same public interface as the underlying subject class, adding a level of indirection by accepting requests from a client object and passing these to the real subject object as necessary.

Behavioural Patterns

The final type of design pattern is the behavioural pattern. Behavioural patterns define manners of communication between classes and objects.

- **Chain of Responsibility**. The chain of responsibility pattern is used to process varied requests, each of which may be dealt with by a different handler.
- **Command**. The command pattern is used to express a request, including the call to be made and all of its required parameters, in a command object. The command may then be executed immediately or held for later use.
- **Interpreter**. The interpreter pattern is used to define the grammar for instructions that form part of a language or notation, whilst allowing the grammar to be easily extended.
- **Iterator**. The iterator pattern is used to provide a standard interface for traversing a collection of items in an aggregate object without the need to understand its underlying structure.

- **Mediator**. The mediator pattern is used to reduce coupling between classes that communicate with each other. Instead of classes communicating directly, and thus requiring knowledge of their implementation, the classes send messages via a mediator object.
- **Memento**. The memento pattern is used to capture the current state of an object and store it in such a manner that it can be restored at a later time without breaking the rules of encapsulation.
- **Observer**. The observer pattern is used to allow an object to publish changes to its state. Other objects subscribe to be immediately notified of any changes.
- **State**. The state pattern is used to alter the behaviour of an object as its internal state changes. The pattern allows the class for an object to apparently change at run-time.
- **Strategy**. The strategy pattern is used to create an interchangeable family of algorithms from which the required process is chosen at run-time.
- **Template Method**. The template method pattern is used to define the basic steps of an algorithm and allow the implementation of the individual steps to be changed.
- **Visitor**. The visitor pattern is used to separate a relatively complex set of structured data classes from the functionality that may be performed upon the data that they hold.

For a quick reference to the design patterns featured in this article, see the Gang of Four Design Patterns Reference Sheet.

ANEXO B – Relación de Anti Patrones

Extraído de: http://www.odi.ch/prog/design/newbies.php (01/12/2014)

Java Anti-Patterns

This page collects some bad code that may not look so obviously bad to beginners. Beginners often struggle with the language syntax. They also have little knowledge about the standard JDK class library and how to make the best use of it. In fact I have collected all examples from everyday junior code. I have modified the original code to give it example character and such that it highlights the problems. Many of these problems can easily be detected by FindBugs, which is available as a simple Eclipse Plug-in. I strongly recommend this tool to any beginner programmer. Also pros should run it from time to time on their codebase, and review its output carefully. It an easy to use tool and I always find some bugs when I use it. An even more complete suite is SonarQube.

Some of these may seem like micro-optimization, premature optimization without profiling or constant factor optimizations. But performance and memory wasted in thousands of these small places adds up quickly and will grind an application to a crawl. And when I say application, I mean a server-side application running on an application server. That's what I do for a living. On desktop GUI applications the situation may not be as bad. But then, what's the only relevant platform that runs client-side Java applications? Android. An embedded platform with very limited resources (memory!). Here even constant factor optimizations pay off quickly. Like iterating over arrays instead of lists.

If you are interested in how to pogram compiler friendly, look at the JDK Performance Wiki.

In the end a lot of your application's performance depends on the overall quality of your code. By the way you should never underestimate the importance of memory footprint. I can't stress that enough. I have seen too many applications with crazy garbage collection overhead and out of memory errors. Even though garbage collection is quite fast, most server-side code's scalability is dominated and limited primarily by its *memory use per request/transaction* and the *request/transaction duration*. Improving either of these by a constant factor will directly give you a higher throughput by that factor. If the factor is 10, it can mean supporting 100 or 1000 users, which can make all the difference to your customer.

Compare these scenarios (assume 100MB young generation):

Scenario	**thread pool**	**tx duration**	**=> max. tx / s**	**mem / tx**	**=> garbage / min**	**GC / min**
base	30	100 ms	**300**	50 KB	**900 MB**	**9**
slower	30	1000 ms	**30**	50 KB	**90 MB**	**0.9**
more mem	30	100 ms	**300**	500 KB	**9 GB**	**90**
excess mem	30	100 ms	**300**	5 MB	**90 GB**	**900**

In the *slower* scenario the transaction duration is 10 times longer. This immediately cuts the maximum number of transactions per second by the factor of 10 as well (limited thread-pool, limited CPU resources). In the *more mem* scenario each transaction uses 10 times as much memory. This directly bumps up the number of garbage collections to over one per second, which causes non-negligible overhead. Using much more memory like in scenario *excess mem* this would lead to 15 collections per second, leaving 66ms per collection which is clearly not enough. The system will thrash. Also 66ms is below the transaction duration of 100ms, so many running transactions will still hold onto memory, preventing it from collection, and causing a propagation of that memory to older generations. This means the older generations will start growing and will need a large (slow) collection sooner. The application in that scenario no longer performs. I think this clearly shows how bad excess memory consumption is, compared to just slow code. All your superfast code can't help you when you allocate too much memory.

- String concatenation
- Lost StringBuffer performance
- Testing for string equality
- Converting numbers to Strings
- Not taking advantage of immutable objects
- XML parsers are for sissies
- Assembling XML with String operations
- The XML encoding trap
- char is not int
- Platform dependent filenames
- Undefined encoding
- Unbuffered streams
- Unbuffered writes to an OutputStreamWriter
- Infinite heap
- Infinite time
- Assuming a cheap timer call
- Catch all: I don't know the right runtime exception
- Exceptions are annoying
- Re-wrapping RuntimeException
- Not properly propagating the exception
- Catching to log
- Incomplete exception handling
- The exception that never happens
- The transient trap
- Overkill initialization
- Log instances: static or not?
- Chosing the wrong class loader
- Poor use of reflection
- Synchronization overkill
- Wrong list type
- The HashMap size trap
- Hashtable, HashMap and HashSet are overrated
- Lists are overrated
- Object arrays are soooo flexible
- Premature object decomposition
- Modifying setters
- Unnecessary Calendar
- Relying on the default TimeZone
- Time zone "conversion"
- Using Calendar.getInstance()
- Dangerous Calendar manipulation
- Calling Date.setTime()
- Assuming SimpleDateFormat was thread-safe
- Having a global Configuration/Parameters/Constants class
- Not noticing overflows
- Using == with float or double
- Storing money in floating point variables

- Not freeing resources in a finally block
- Abusing finalize()
- Involuntarily resetting Thread.interrupted
- Spawning thread from static initializers
- Canceled timer tasks that keep state
- Holding strong references to ClassLoaders and unflushable caches

String concatenation

```
String s = "";
for (Person p : persons) {
   s += ", " + p.getName();
}
s = s.substring(2); //remove first comma
```

This is a real performance killer: O(persons.length²). The repeated concatenation of strings in a loop causes excess garbage and array copying. Moreover it is ugly that the resulting string has to be fixed for an extra comma.

```
StringBuilder sb = new StringBuilder(persons.size() * 16); // well estimated buffer
for (Person p : persons) {
   if (sb.length() > 0) sb.append(", ");
   sb.append(p.getName);
}
```

Lost StringBuffer performance

```
StringBuffer sb = new StringBuffer();
sb.append("Name: ");
sb.append(name + '\n');
sb.append("!");
...
String s = sb.toString();
```

Despite good intentions the above code is not perfect. The most obvious mistake is the string concatenation in line 3. In line 4 appending a char would be faster than appending a String. An also major omission is the missing length initialization of the buffer which may incur unnecessary resizing (array copying). In JDK 1.5 and above a StringBuilder instead of StringBuffer should have been used: because it is only a local variable the implicit synchronization is overkill. Actually, using simple String concatenation compiles to almost perfect byte code: it's only missing the length initialization.

```
StringBuilder sb = new StringBuilder(100);
sb.append("Name: ");
sb.append(name);
sb.append("\n!");
String s = sb.toString();
String s = "Name: " + name + "\n!";
```

Testing for string equality

```
if (name.compareTo("John") == 0) ...
if (name == "John") ...
if (name.equals("John")) ...
if ("".equals(name)) ...
```

None of the above comparisons is wrong - but neither are they really good. The compareTo method is overkill and too verbose. The == operator tests for object identity which is probably not what you want. The equals method is the way to go, but reversing the constant and variable would give you extra safety if name is null.

```
if ("John".equals(name)) ...
if (name.length() == 0) ...
if (name.isEmpty()) ...
```

Converting numbers to Strings

```
"" + set.size()
new Integer(set.size()).toString()
```

The return type of the Set.size() method is int. A conversion to String is wanted. These two examples in fact do the conversion. But the first incurs the penalty of a concatenation operation (translates to (new StringBuilder()).append(i).toString())). And the second creates an intermediate Integer wrapper. The correct way of doing it is one of these

```
Integer.toString(set.size())
```

Not taking advantage of immutable objects

```
zero = new Integer(0);
return Boolean.valueOf("true");
```

Integer as well as Boolean are immutable. Thus it doesn't make sense to create several objects that represent the same value. Those classes have built-in caches for frequently used instances. In the case of Boolean there are even only two possible instances. The programmer can take advantage of this:

```
zero = Integer.valueOf(0);
return Boolean.TRUE;
```

XML parsers are for sissies

```
int start = xml.indexOf("<name>") + "<name>".length();
int end = xml.indexOf("</name>");
String name = xml.substring(start, end);
```

This naive XML parsing only works with the most simple XML documents. It will however fail if a) the name element is not unique in the document, b) the content of name is not only character data c) the text data of name contains escaped characters d) the text data is specified as a CDATA section e) the document uses XML namespaces. XML is way too complex for string operations. There is a reason why XML parsers like Xerces are a over one megabyte jar files! The equivalent with JDOM is:

```
SAXBuilder builder = new SAXBuilder(false);
Document doc = doc = builder.build(new StringReader(xml));
String name = doc.getRootElement().getChild("name").getText();
```

Assembling XML with String operations

```
String name = ...
String attribute = ...
String xml = "<root>"
        +"<name att=\""+ attribute +"\">"+ name +"</name>"
        +"</root>";
```

Many beginners are tempted to produce XML output like shown above, by using String operations (which they know so well and which are so easy). Indeed it is very simple and almost beautiful code. However it has one severe shortcoming: It fails to escape reserved characters. So if the variables name or attribute contain any of the reserved characters <, >, &, " or ' this code would produce invalid XML. Also as soon as the XML uses namespaces, String operations may quickly become nasty and hard to maintain. Now XML should be assembled in a DOM. The JDom library is quite nice for that.

```
Element root = new Element("root");
root.setAttribute("att", attribute);
root.setText(name);
Document doc = new Documet();
doc.setRootElement(root);
XmlOutputter out = new XmlOutputter(Format.getPrettyFormat());
String xml = out.outputString(root);
```

The XML encoding trap

```
String xml = FileUtils.readTextFile("my.xml");
```

It is a very bad idea to read an XML file and store it in a String. An XML specifies its encoding in the XML header. But when reading a file you have to know the encoding beforehand! Also storing an XML file in a String wastes memory. All XML parsers accept an InputStream as a parsing source and they figure out the encoding themselves correctly. So you can feed them an InputStream instead of storing the whole file in memory temporarily. The byte order (big-endian, little-endian) is another trap when a multi-byte encoding (such as UTF-8) is used. XML files may carry a byte order mark at the beginning that specifies the byte order. XML parsers handle them correctly.

char is not int

```
int i = in.read();
char c = (char) i;
```

The above code assumes that you can create a character from a number. It's true technically: a character's number is the 16 bit Unicode codepoint number. But it is semantic nonsense. In Java the character is a semantic entity of its own. The character's byte representation is completely decoupled from that. If we encounter a char we don't need to worry whether the character is stored in UTF-8, UTF-16, USC-4 or ISO-8859-1 internally. It simply doesn't matter. We can compare it to other characters and it will always behave as expected. This concept is not known in C for example. In C the char type is just a numeric type. It can contain anything, even invalid data that does not represent characters. In C you have to know exactly which character encoding a char array uses or you may do wrong things when sorting, printing, searching etc. Also C programs may wrongly assume that a char is one byte long and contains values 0-127 or 0-256, which is true for ASCII, but not for many other character encodings (known as "multi-byte" character encodings). Anyway, in Java use Reader/Writer or CharsetEncoder/CharsetDecoder instead to convert between characters and their byte representation (see following paragraph).

Platform dependent filenames

```
File tmp = new File("C:\\Temp\\1.tmp");
File exp = new File("export-2013-02-01T12:30.txt");
File f = new File(path +'/'+ filename);
```

Never hard code paths in a filesystem. Different platforms have different conventions, and you can never be sure that a hard coded path is actually available on a random system. Use API calls to create temporary files. Mind that different file systems have different restrictions on what makes a valid file name. Here the exp file contains a colon character, which is illegal on Windows file systems. When you construct absolute or relative paths in the filesystem, be careful of the platform dependent separator character.

```
File tmp = File.createTempFile("myapp","tmp");
File exp = new File("export-2013-02-01_1230.txt");

File f = new File(path + File.separatorChar + filename);
// or even better
File dir = new File(path);
File f = new File(dir, filename);
```

Undefined encoding

```
Reader r = new FileReader(file);
Writer w = new FileWriter(file);
Reader r = new InputStreamReader(inputStream);
Writer w = new OutputStreamWriter(outputStream);
String s = new String(byteArray); // byteArray is a byte[]
byte[] a = string.getBytes();
```

Each line of the above converts between byte and char using the default platform encoding. The code behaves differently depending on the platform it runs on. This is harmful if the data flows from one platform to another. It is considered bad practice to rely on the default platform encoding at all. Conversions should always be performed with a defined encoding.

```
Reader r = new InputStreamReader(new FileInputStream(file), "ISO-8859-1");
Writer w = new OutputStreamWriter(new FileOutputStream(file), "ISO-8859-1");
Reader r = new InputStreamReader(inputStream, "UTF-8");
Writer w = new OutputStreamWriter(outputStream, "UTF-8");
String s = new String(byteArray, "ASCII");
byte[] a = string.getBytes("ASCII");
```

Unbuffered streams

```
InputStream in = new FileInputStream(file);
int b;
while ((b = in.read()) != -1) {
   ...
}
```

The above code reads a file byte by byte. Every read() call on the stream will cause a native (JNI) call to the native implementation of the filesystem. Depending on the implementation this may cause a syscall to the operating system. JNI calls are expensive and so are syscalls. The number of native calls can be reduced dramatically by wrapping the stream into a BufferedInputStream. Reading 1 MB of data from /dev/zero with the above code took about 1 second on my laptop. With the fixed code below it was down to 60 milliseconds! That's a 94% saving. This also applies for output streams of course. And it is true not only for the file system but also for sockets.

```
InputStream in = new BufferedInputStream(new FileInputStream(file));
```

Unbuffered writes to an OutputStreamWriter

```
Writer w = new OutputStreamWriter(os, "UTF-8");
while (...) {
  w.write("something");
}
```

As demonstrated OutputStreamWriter uses memory for each call to its write() methods. This is very unfortunate and not the behaviour that one would expect! If you do many writes, you should wrap it in a BufferedWriter, which (also unexpectedly) seems to use no memory at all:

```
Writer w = new BufferedWriter(new OutputStreamWriter(os, "UTF-8"));
```

Infinite heap

```
byte[] pdf = toPdf(file);
```

Here a method creates a PDF file from some input and returns the binary PDF data as a byte array. This code assumes that the generated file is small enough to fit into the available heap memory. If this code can not make this 100% sure then it is vulnerable to an out of memory condition. Especially if this code is run server-side which usually means many parallel threads. Bulk data must never be handled with byte arrays. Streams should be used and the data should be spooled to disk or a database.

```
File pdf = toPdf(file);
```

A similar anti-pattern is to buffer streaming input from an "untrusted" (security term) source. Such as buffering data that arrives on a network socket. If the application doesn't know how much data will be arriving it must make sure that it keeps an eye on the size of the data. If the amount of buffered data exceeds sane limits an error condition (exception) should be signalled to the caller, rather than driving the application against the wall by letting it run into an out of memory condition.

Infinite time

```
Socket socket = ...
socket.connect(remote);
InputStream in = socket.getInputStream();
int i = in.read();
```

The above code has two blocking calls that use unspecified timeouts. Imagine if the timeout is infinite. That may cause the application to hang forever. Generally it is an extremely stupid idea to have infinite timeouts in the first place. Infinity is extremely long. Even by the time the Sun turns into a red giant (it explodes), it's still a looong way to Infinity. The average programmer dies at 72. There is simply **no** real-world situation, where we want to wait that long. Infinite timeout is just an absurd thing. Use an hour, day, week, month, 1 year, 10 years. But not Infinity. To connect to a remote machine I personally find 20 seconds plenty of timeout. A human is not even as patient and would cancel the operation before. While there is a nice override for the connect() method that takes a timeout parameter, there is no such thing for the read(). But you can modify a Socket's socket timeout before every blocking call. (Not just once! You can set different timeouts for different situations.) The socket will throw an exception on blocking calls after that timeout. Also frameworks that communicate over the network should provide an API to control these timeouts and use sensible default values. Infinity is not sensible - it's insane and drives you mad. Who came up with this absolutely useless infinity timeout anyway?

```
Socket socket = ...
socket.connect(remote, 20000); // fail after 20s
InputStream in = socket.getInputStream();
socket.setSoTimeout(15000);
int i = in.read();
```

Unfortunately the file system API (FileInputStream, FileChannel, FileDescriptor, File) provides no way to set timeouts on file operations. That's very unfortunate. Because these are the most common blocking calls in a Java application: writing to stdout/stderr and reading from stdin are file operations, and writing to log files is common. Operations on the standard input/output streams depend directly on other processes outside of our Java VM. If they decide to block forever, so will reads/writes to these streams in our application. Disk I/O is a limited resource for which all processes on a system compete. There is no guarantee that a simple read/write on a file is quick. It may incur unspecified wait time. Also today remote file systems are ubiquitous. Disks may be on a SAN/NAS, or file systems may be mounted over the network (NFS, AFS, CIFS/Samba). So a filesystem call may actually be a network call: too bad that we don't have the power of the network API here! So if the OS decides that the timeout for the write is 60 seconds you're stuck with it. It is a failure to assume that any disk/file operation is fast, or even remotely instantaneous. An application can do the user a favour by assuming that a file operation can takes seconds. So it's best avoided or done asynchronously (in background). Solutions to this problem are: adequate buffering and queueing/asynchronous processing.

Assuming a cheap timer call

```
for (...) {
  long t = System.currentTimeMillis();
  long t = System.nanoTime();
  Date d = new Date();
  Calendar c = new GregorianCalendar();
}
```

Creating a new Date or Calendar performs a syscall to obtain the current time. On Unix/Linux this is the syscall gettimeofday which is considered "extremely cheap". Well, extremely cheap only compared to other syscalls! In that it usually doesn't require a switch from userspace to kernelspace but is rather implemented as a read from a memory mapped page. Still calls to gettimeofday are expensive compared to normal code execution. The exact penalty of the call strongly depends on the architecture and even configuration (modern x86 systems have numerous timers that can be used by the OS: HPET, TSC, RTC, ACPI, clock chips etc.). On my Linux-2.6.37-rc7 system the timer calls also seem to be synchronised over the system. That means the total available bandwidth of ~800 calls per ms is shared by all threads/processes. Consequently my dual core running with 2 threads was able to make ~400 calls per ms per thread. (Thanks to J. Davies for that hint) And last but not least the resolution of this timer is not infinite. At best it is milliseconds, but it may well be rather something like 25 to 50 milliseconds with a large jitter. Modern Linux system can easily achieve the full ms resolution in System.currentTimeMillis. But that has not always been the case. System.nanoTime will certainly not have its full theoretical resolution: 1ns = 10^{-9}s which corresponds to 1GHz. So on a CPU with 3GHz this would allow ~3 instructions to execute the call, which is obviously not enough. I measured a large jitter between 800ns and 1000000ns(1ms). Clearly calling gettimeofday every 100 nano seconds is wasteful.

Most of the time you don't need the current time as precicely. Caching it outside of the loop is trivial. This way you only access the timer once. You can still decide to clone the Date instance, if you really need different objects. Cloning is extremely cheap compared to a timer access (factor 50 on my system).

```
Date d = new Date();
for (E entity : entities) {
  entity.doSomething();
  entity.setUpdated((Date) d.clone());
}
```

Caching the time may not be an option if the loop runs for more than a couple of milliseconds. In that case you may setup a timer that periodically updates a timestamp variable with the current time (using interrupts). Set it to the exact granularity that you need. The coarser that granularity is, the better. On my system this loop is 200 times faster than creating a new Date each time.

```
private volatile long time;

Timer timer = new Timer(true);
try {
  time = System.currentTimeMillis();
  timer.scheduleAtFixedRate(new TimerTask() {
    public void run() {
      time = System.currentTimeMillis();
    }
  }, 0L, 10L); // granularity 10ms
  for (E entity : entities) {
     entity.doSomething();
     entity.setUpdated(new Date(time));
  }
} finally {
  timer.cancel();
}
```

Catch all: I don't know the right runtime exception

```
Query q = ...
Person p;
try {
   p = (Person) q.getSingleResult();
} catch(Exception e) {
   p = null;
}
```

This is an example of a J2EE EJB3 query. The getSingleResult throws runtime exceptions when a) the result is not unique, b) there is no result c) when the query could not be executed due to database failure or so. The code above just catches any exception. A typical catch-all block. Using null as a result may be the right thing for case b) but not for case a) or c). In general one should not catch more exceptions than necessary. The correct exception handling is

```
Query q = ...
Person p;
try {
   p = (Person) q.getSingleResult();
} catch(NoResultException e) {
   p = null;
}
```

Exceptions are annoying

```
try {
   doStuff();
} catch(Exception e) {
   log.fatal("Could not do stuff");
}
doMoreStuff();
```

There are two problems with this tiny piece of code. First, if this is really a fatal condition then the method should abort and notify the caller of the fatal condition with an appropriate exception (so why is it caught in the first place?) Hardly ever can you just continue after a fatal condition. Second, this code is very hard to debug because the reason of the failure is lost. Exception objects carry detailed information about where the error occurred and what caused it. Individual subclasses may actually carry a lot of extra information that the caller can use to deal with the situation properly. It's a lot more than a simple error code (which is so popular in the C world. Just look at the Linux kernel. return -EINVAL everywhere...). If you catch highlevel exceptions then at least log the message and stack trace. You should not see exceptions as a necessary evil. They are a great tool for error handling.

```
try {
   doStuff();
} catch(Exception e) {
    throw new MyRuntimeException("Could not do stuff because: "+ e.getMessage(), e);
}
```

Re-wrapping RuntimeException

```
try {
  doStuff();
} catch(Exception e) {
  throw new RuntimeException(e);
}
```

Sometimes you really want to re-throw any checked exception as RuntimeException. The above piece of code doesn't take into account however, that RuntimeException extends Exception. The RuntimeException doesn't need to be catched here. Also the exception's message is not propagated properly. A bit better is to catch the RuntimeException separately and not wrap it. Even better is to catch all the checked exceptions individually (even if they are a lot).

```
try {
  doStuff();
} catch(RuntimeException e) {
  throw e;
} catch(Exception e) {
  throw new RuntimeException(e.getMessage(), e);
}
try {
  doStuff();
} catch(IOException e) {
  throw new RuntimeException(e.getMessage(), e);
} catch(NamingException e) {
  throw new RuntimeException(e.getMessage(), e);
}
```

Not properly propagating the exception

```
try {
} catch(ParseException e) {
  throw new RuntimeException();
  throw new RuntimeException(e.toString());
  throw new RuntimeException(e.getMessage());
  throw new RuntimeException(e);
}
```

This codes just wraps a parsing error into a runtime exception in different ways. None of them provides really good information to the caller. The first just loses all information. The second may do anything depending on what information toString() produces. The default toString() implementation lists the fully qualified exception name followed by the message. Nesting many exceptions will produce an unwieldy long and ugly string, unsuitable for a user. The third just preserves the message, which is better than nothing. The last preserves the cause, but sets the message of the runtime exception to toString() of its cause (see above). The most useful and readable version is to propagate only the cause message in the runtime exception and pass the original exception as the cause:

```
try {
} catch(ParseException e) {
  throw new RuntimeException(e.getMessage(), e);
}
```

Catching to log

```
try {
    ...
} catch(ExceptionA e) {
   log.error(e.getMessage(), e);
   throw e;
} catch(ExceptionB e) {
   log.error(e.getMessage(), e);
   throw e;
}
```

This code only catches exception to write out a log statement and then rethrows the same exception. This is stupid. Let the caller decide if the message is important to log and remove the whole try/catch clause. Its only useful when you know that the caller doesn't log it. That's the case if the method is called by a framework which is not under your control.

Incomplete exception handling

```
try {
   is = new FileInputStream(inFile);
   os = new FileOutputStream(outFile);
} finally {
   try {
      is.close();
      os.close();
   } catch(IOException e) {
      /* we can't do anything */
   }
}
```

If streams are not closed, the underlying operating system can't free native resources. This programmer wanted to be careful about closing both streams. So he put the close in a finally clause. But if is.close() throws an IOException then os.close is not even executed. Both close statements must be wrapped in their own try/catch clause. Moreover, if creating the input stream throws an exception (because the file was not found) then os is null and os.close() will throw a NullPointerException. To make this less verbose I have stripped some newlines.

```
try {
   is = new FileInputStream(inFile);
   os = new FileOutputStream(outFile);
} finally {
   try { if (is != null) is.close(); } catch(IOException e) {/* we can't do anything */}
   try { if (os != null) os.close(); } catch(IOException e) {/* we can't do anything */}
}
```

The exception that never happens

```
try {
  ... do risky stuff ...
} catch(SomeException e) {
  // never happens
}
... do some more ...
```

Here the developer executes some code in a try/catch block. He doesn't want to rethrow the exception that one of the called methods declares to his annoyance. As the developer is clever he knows that in his particular situation the exception will never be thrown, so he just inserts an empty catch block. He even puts a nice comment in the empty catch block - but they are famous last words... The problem with this is: how can he be sure? What if the implementation of the called method changes? What if the exception is still thrown in some special case but he just didn't think of it? The code after the try/catch may do the wrong thing in that situation. The exception will go completely unnoticed. The code can be made much more reliable by throwing a runtime exception in the case. This works like an assertion and adheres to the "crash early" principle. The developer will notice if his assumption was wrong. The code after the try/catch will not be executed if the exception occurred against all honest hope and expectation. If the exception really never occurs - fine, nothing changed.

```
try {
  ... do risky stuff ...
} catch(SomeException e) {
  // never happens hopefully
  throw new IllegalStateException(e.getMessage(), e); // crash early, passing all information
}
... do some more ...
```

The transient trap

```
public class A implements Serializable {
   private String someState;
   private transient Log log = LogFactory.getLog(getClass());

   public void f() {
      log.debug("enter f");
      ...
   }
}
```

Log objects are not serializable. The programmer knew this and correctly declared the log field as transient so it is not serialised. However the initialisation of this variables happens in the class' initialiser. Upon deserialization initializers and constructors are not executed! This leaves the deserialized object with a null log variable which subsequently causes a NullPointerException in f(). Rule of thumb: never use class initialization with transient variables. You can either solve this case here by using a static variable or by using a local variable:

```
public class A implements Serializable {
   private String someState;
   private static final Log log = LogFactory.getLog(A.class);

   public void f() {
      log.debug("enter f");
      ...
   }
}

public class A implements Serializable {
   private String someState;

   public void f() {
      Log log = LogFactory.getLog(getClass());
      log.debug("enter f");
      ...
   }
}
```

Overkill initialization

```
public class B {
    private int count = 0;
    private String name = null;
    private boolean important = false;
}
```

This programmer used to code in C. So naturally he wants to make sure every variable is properly initialized. Here however it is not necessary. The Java language specification guarantees that member variables are initialized with certain values automatically: 0, null, false. By declaring them explicitly the programmer causes a class initializer to be executed before the constructor. This is unnecessary overkill and should be avoided.

```
public class B {
    private int count;
    private String name;
    private boolean important;
}
```

Log instances: static or not?

This section was edited and before actually suggested not to store log instances in static variables. Turns out I was wrong. Mea culpa. I apologize.
Store the darn log instance in a static final variable and be happy.

```
private static final Log log = LogFactory.getLog(MyClass.class);
```

Here is why:

- Automatically thread-safe. But only with the final keyword included!
- Usable from static and non-static code.
- No problems with serializable classes.
- Initialization cost only once: getLog() may not be as cheap as you might suppose.
- Nobody is going to unload the Log class loader anyway.

Chosing the wrong class loader

```
Class clazz = Class.forName(name);
Class clazz = getClass().getClassLoader().loadClass(name);
```

This code uses the class loader that loaded the current class. getClass() might return something unexpected, like a subclass, or a dynamic proxy. Something out of your control. This is hardly ever what you want when you dynamically load an additional class. Especially in managed environments like Application servers, Servlet engines or Java Webstart this is most certainly wrong. This code will behave very differently depending on the environment it is run in. Environments use the context class loader to provide applications with a class loader they should use to retrieve "their own" classes.

```
ClassLoader cl = Thread.currentThread().getContextClassLoader();
if (cl == null) cl = MyClass.class.getClassLoader(); // fallback
Class clazz = cl.loadClass(name);
```

Poor use of reflection

```
Class beanClass = ...
if (beanClass.newInstance() instanceof TestBean) ...
```

This programmer is struggling with the reflection API. He needs a way to check for inheritance but didn't find a way to do it. So he just creates a new instance and uses the instanceof operator he is used to. Creating an instance of a class you don't know is dangerous. You never know what this class does. It may be very expensive. Or the default constructor may not even exist.

Then this if statement would throw an exception. The correct way of doing this check is to use the Class.isAssignableFrom(Class) method. Its semantics is upsidedown of instanceof.

```
Class beanClass = ...
if (TestBean.class.isAssignableFrom(beanClass)) ...
```

Synchronization overkill

```
Collection l = new Vector();
for (...) {
  l.add(object);
}
```

Vector is a synchronized ArrayList. And Hashtable is a synchronized HashMap. Both classes should only be used if synchronization is explicitly required. If however those collections are used as local temporary variables the synchronization is complete overkill and degrades performance considerably. I measured a 25% penalty.

```
Collection l = new ArrayList();
for (...) {
  l.add(object);
}
```

Wrong list type

Without sample code. Junior developers often have difficulties to chose the right list type. They usually choose quite randomly from Vector, ArrayList and LinkedList. But there are performance considerations to make! The implementations behave quite differently when adding, iterating or accessing object by index. I'll ignore Vector in this list because it behaves like an ArrayList, just slower. NB: n is the size of the list, not the number of operations!

	ArrayList	**LinkedList**
add (append)	O(1) or ~O(log(n)) if growing	O(1)
insert (middle)	O(n) or ~O(n*log(n)) if growing	O(n)
remove (middle)	O(n) (always performs complete copy)	O(n)
iterate	O(n)	O(n)
get by index	O(1)	O(n)

The insert performance of the ArrayList depends on whether it has to grow during the insert or if the initial size is reasonably set. The growing occurs exponentially (by factor 2) so growing costs are O(log(n)). The exponential growing however may use much more memory than you actually need. The sudden need to resize the list also makes the response time sluggisch and will probably cause a major garbage collection if the list is large. Iterating over the lists is equally inexpensive. Indexed list element access however is very slow in linked lists of course. Memory considerations: LinkedList wraps every element into a wrapper object. ArrayList allocates a completely new array each time it needs to grow and performs an array copy on every remove(). All standard Collections can not reuse their Iterator objects, which may cause iterator churn especially when recursively iterating large tree structures. Personally I almost never use LinkedList. It would really only make sense when you wanted to insert objects in the middle of a list. But without access to the wrapper object this doesn't scale with O(1) but O(n) because you must first traverse the list until you find the insert position. So what exactly is the point of the LinkedList class? I recommend using ArrayLists only.

The HashMap size trap

```
Map map = new HashMap(collection.size());
for (Object o : collection) {
  map.put(o.key, o.value);
}
```

This developer had good intentions and wanted to make sure that the HashMap doesn't need to be resized. He thus set its initial size to the number of elements he was going to put into it. Unfortunately the HashMap implementation doesn't quite behave like this. It sets its internal threshold to threshold = (int)(capacity * loadFactor). So it will resize after 75% of the collection have been inserted into the map. The above code will thus always cause extra garbage.

```
Map map = new HashMap(1 + (int) (collection.size() / 0.75));
```

Hashtable, HashMap and HashSet are overrated

These classes are extremely popular. Because they have great usability for the developer. Unfortunately they are also horribly inefficient. Hashtable and HashMap wrap every key/value pair into an Entry wrapper object. An Entry object is surprisingly large. Not only does it hold a reference to key and value, but also stores the hash code and a forward reference to the next Entry of the hash bucket. When you look at heap dumps with a memory analyzer you will be shocked by how much space is wasted by them in large applications like an application server. When you look at the source code of HashSet you will see that the developers were extremely lazy and just used a HashMap in the backend!
Before using any of these classes, think again. IdentityHashMap can be a viable alternative. But be careful, it intentionally breaks the Map interface. It is much more memory efficient by implementing an open hashtable (no buckets), doesn't need an Entry wrapper and uses a simple Object[] as its backend. Instead of a HashSet a simple ArrayList may do similarly well (you can use contains(Object)) as long as it's small and lookups are rare.
For Sets that contain only a handful of entries the whole hashing is overkill and the memory wasted for the HashMap backend plus the wrapper objects is just nuts. Just use an ArrayList or even an array.
Actually it's a shame that there is no efficient Map and Set implementations in the standard JDK!

Lists are overrated

Also List implementations are very popular. But even lists are often not necessary. Simple arrays may do as well. I am not saying that you should not use Lists at all. They are great to work with. But know when to use arrays. The following are indicators that you should be using an array instead of a list:

- The list has a fixed size. Example: days of the week. A set of constants.
- The list is often (10'000 times) traversed.
- The list contains wrapper objects for numbers (there are no lists of primitive types).

Let me illustrate that in code:

```
List<Integer> codes = new ArrayList<Integer>();
codes.add(Integer.valueOf(10));
codes.add(Integer.valueOf(20));
codes.add(Integer.valueOf(30));
codes.add(Integer.valueOf(40));
```

versus

```
int[] codes = { 10, 20, 30, 40 };
// horribly slow and a memory waster if l has a few thousand elements (try it yourself!)
List<Mergeable> l = ...;
for (int i=0; i < l.size()-1; i++) {
   Mergeable one = l.get(i);
   Iterator<Mergeable> j = l.iterator(i+1); // memory allocation!
   while (j.hasNext()) {
        Mergeable other = l.next();
        if (one.canMergeWith(other)) {
            one.merge(other);
            other.remove();
        }
    }
}
```

versus

```
// quite fast and no memory allocation
Mergeable[] l = ...;
for (int i=0; i < l.length-1; i++) {
    Mergeable one = l[i];
    for (int j=i+1; j < l.length; j++) {
        Mergeable other = l[j];
        if (one.canMergeWith(other)) {
            one.merge(other);
            l[j] = null;
        }
    }
}
```

You save an extra list object (wrapping an array), wrapper objects and possibly lots of iterator instances. Even Sun realized this. That's why Collections.sort() actually copies the list into an array and performs the sort on the array.

Object arrays are soooo flexible

```
/**
 * @returns [1]: Location, [2]: Customer, [3]: Incident
 */
Object[] getDetails(int id) {...
```

Even though documented, this kind of passing back values from a method is ugly and error prone. You should really declare a small class that holds the objects together. This is analoguos to a struct in C.

```
Details getDetails(int id) {...}

private class Details {
    public Location location;
    public Customer customer;
    public Incident incident;
}
```

Premature object decomposition

```
public void notify(Person p) {
    ...
    sendMail(p.getName(), p.getFirstName(), p.getEmail());
    ...
}
class PhoneBook {
    String lookup(String employeeId) {
        Employee emp = ...
        return emp.getPhone();
    }
}
```

In the first example it's painful to decompose an object just to pass its state on to a method. In the second example the use of this method is very limited. If overall design allows it pass the object itself.

```
public void notify(Person p) {
   ...
   sendMail(p);
   ...
}
class EmployeeDirectory {
   Employee lookup(String employeeId) {
      Employee emp = ...
      return emp;
   }
}
```

Modifying setters

```
private String name;

public void setName(String name) {
   this.name = name.trim();
}

public void String getName() {
   return this.name;
}
```

This poor developer suffered from spaces at the beginning or end of a name entered by the user. He thought to be clever and just removed the spaces inside the setter method of a bean. But how odd is a bean that modifies its data instead of just holding it? Now the getter returns different data than was set by the setter! If this was done inside an EJB3 entity bean a simple read from the DB would actually modify the data: For every INSERT there would be an UPDATE statement. Let alone how hard it is to debug these side-effects! In general, a bean should not modify its data. It is a data container, not business logic. Do the trimming where it makes sense: in the controller where the input occurs or in the logic where the spaces are not wanted.

```
person.setName(textInput.getText().trim());
```

Unnecessary Calendar

```
Calendar cal = new GregorianCalender(TimeZone.getTimeZone("Europe/Zurich"));
cal.setTime(date);
cal.add(Calendar.HOUR_OF_DAY, 8);
date = cal.getTime();
```

A typical mistake by a developer who is confused about date, time, calendars and time zones. To add 8 hours to a Date there is no need for a Calendar. Neither is the time zone of any relevance. (Think about is if you don't understand this!) However if we wanted to add days (not hours) we would need a Calendar, because we don't know the length of a day for sure (on DST change days may have 23 or 25 hours).

```
date = new Date(date.getTime() + 8L * 3600L * 1000L); // add 8 hrs
Calendar cal = new GregorianCalender(TimeZone.getTimeZone("Europe/Zurich"));
SimpleDateFormat df = new SimpleDateFormat("dd.MM.yyyy HH:mm");
df.setCalendar(cal);
```

Here the Calendar object is completely unnecessary. The DateFormat object already contains a Calendar instance. Reuse that.

```
SimpleDateFormat df = new SimpleDateFormat("dd.MM.yyyy HH:mm");
df.setTimeZone(TimeZone.getTimeZone("Europe/Zurich"));
```

Relying on the default TimeZone

```
Calendar cal = new GregorianCalendar();
cal.setTime(date);
cal.set(Calendar.HOUR_OF_DAY, 0);
cal.set(Calendar.MINUTE, 0);
cal.set(Calendar.SECOND, 0);
Date startOfDay = cal.getTime();
```

The developer wanted to calculate the start of the day (0h00). First he obviously missed out the millisecond field of the Calendar. But the real big mistake is not setting the TimeZone of the Calendar object. The Calendar will thus use the default time zone. This may be fine in a Desktop application, but in server-side code this is hardly ever what you want: 0h00 in Shanghai is in a very different moment than in London. The developer needs to check which is the time zone that is relevant for this computation.

```
Calendar cal = new GregorianCalendar(user.getTimeZone());
cal.setTime(date);
cal.set(Calendar.HOUR_OF_DAY, 0);
cal.set(Calendar.MINUTE, 0);
cal.set(Calendar.SECOND, 0);
cal.set(Calendar.MILLISECOND, 0);
Date startOfDay = cal.getTime();
```

Time zone "conversion"

```
public static Date convertTz(Date date, TimeZone tz) {
  Calendar cal = Calendar.getInstance();
  cal.setTimeZone(TimeZone.getTimeZone("UTC"));
  cal.setTime(date);
  cal.setTimeZone(tz);
  return cal.getTime();
}
```

If you think this method does something useful, please go and read the article about time. This developer had not read the article and was desperately trying to "fix" the time zone of his date. Actually the method does nothing. The returned Date will not have any different value than the input. Because a Date does not carry time zone information. It is always UTC. And the getTime / setTime methods of Calendar always convert between UTC and the actual time zone of the Calendar.

Using Calendar.getInstance()

```
Calendar c = Calendar.getInstance();
c.set(2009, Calendar.JANUARY, 15);
```

This code assumes a Gregorian calendar. But what if the returned Calendar subclass is a Buddhistic, Julian, Hebrew, Islamic, Iranian or Discordian calendar? In these the year 2009 has a very different meaning. And a month called January doesn't exist. Calendar.getInstance() uses the current default locale to select an appropriate implementation. It depends on the Java implementaton which implementations are available. The utility of Calendar.getInstance() is thus very limited, and its use should be avoided as it's result is not well defined.

```
Calendar c = new GregorianCalendar(timeZone);
c.set(2009, Calendar.JANUARY, 15);
```

Dangerous Calendar manipulation

```
GregorianCalender cal = new GregorianCalender(TimeZone.getTimeZone("Europe/Zurich"));
cal.set(Calendar.SECOND, 0);
cal.set(Calendar.MILLISECOND, 0);
if (cal.before(other)) doSomething();

cal.setTimeZone(TimeZone.getTimeZone("GMT"));
cal.set(Calendar.HOUR_OF_DAY, 23);
Date d = cal.getTime();
```

This code manipulates a Calendar object in ways that are bound to yield undefined results. Calendar objects have complex inner state: individual fields for day, hour, year etc., a millisecond since epoch value (like Date) and a time zone. Depending on what you change, some of these fields are invalidated and are only recomputed from other values when you call certain methods:

- set() invalidates the millisecond since epoch value and dependent fields (changing DATE obviously invalidates DAY_OF_WEEK)
- setTimeZone() invalidates all fields execpt the millisecond since epoch value
- get(), getTime(), getTimeInMillis(), add(), roll() recomputes the millisecond since epoch value from the fields
- get(), add() also recompute invalid fields from millisecond since epoch

Whenever you change fields with set(), then dependend fields do not get updated until you call get(), getTime(), getTimeInMillis(), add(), or roll(). The first paragraph of above code calls set() followed by before(). There is no guarantee (according to the API Doc) that before() will see the modified time value.

The second paragraph invalidates all fields and the millisecond since epoch value by calling setTimeZone() and set(), losing the calendar's data completely. See also bug 4827490

Calendar objects should always be manipulated according to these simple rules:

- Initialize TimeZone (and Locale if you need) already in the constructor
- After calls to set() add a call to getTimeInMillis()
- After a call to setTimeZone() add a call to get()

```
GregorianCalender cal = new GregorianCalender(TimeZone.getTimeZone("Europe/Zurich"));
cal.set(Calendar.SECOND, 0);
cal.set(Calendar.MILLISECOND, 0);
cal.getTimeInMillis();
if (cal.before(other)) doSomething();

cal.setTimeZone(TimeZone.getTimeZone("GMT"));
cal.get(Calendar.DATE);
cal.set(Calendar.HOUR_OF_DAY, 23);
Date d = cal.getTime();
```

Calling Date.setTime()

```
account.changePassword(oldPass, newPass);
Date lastmod = account.getLastModified();
lastmod.setTime(System.currentTimeMillis());
```

The above code updates the last modified date of the account entity. The programmer wants to be conservative and avoids creating a new Date object. Instead she uses the the setTime method to modify the existing Date instance.

There is actually nothing wrong with that. But I just do not recommend this practice. Date objects are usually passed around carelessly. The same Date instance could be passed to numerous objects, which don't make a copy in their setters. Dates are often used like primitives. Thus if you modify a Date instance, other objects that use this instance might behave unexpectedly. Of course it is unclean design if an object exposes its intrinsic Date instance to the outside world, if you write code that strictly adheres to classical OO-principles (which I think is too inconvenient). General everyday Java practice however is to just copy Date references and not clone the object in setters. Thus every programmer should treat Date as immutable and should not modify existing instances. This should only be done for performance reasons in special situations. Even then the use of a simple long is probably equally good.

```
account.changePassword(oldPass, newPass);
account.setLastModified(new Date());
```

Assuming SimpleDateFormat was thread-safe

```
public class Constants {
   public static final SimpleDateFormat date = new SimpleDateFormat("dd.MM.yyyy");
}
```

The above code is flawed in several ways. It's broken, because it shares a static instance of a SimpleDateFormat with possibly any number of threads. SimpleDateFormat is not thread-safe. If multiple threads concurrently use this object the results are undefined. You may observe strange output from format and parse or even exceptions. Unfortunately this mistake is very common!

Yes, sharing a SimpleDateFormat requires proper synchronization. Yes that comes at a price (cache flushes, lock contention, etc.). And yes, creating a SimpleDateFormat is not free either (pattern parsing, object allocation). But simply ignoring thread-safety is not a solution, but a sure way to break your code.

Of course this code also doesn't take the time zone into account. And then defining a class called Constants screams of yet another anti-pattern (see next section).

Having a global Configuration/Parameters/Constants class

```
public interface Constants {
   String version = "1.0";
   String dateFormat = "dd.MM.yyyy";
   String configFile = ".apprc";
   int maxNameLength = 32;
   String someQuery = "SELECT * FROM ...";
}
```

Often seen in large projects: one class or interface that contains all sorts of constants that are used throughout the application. Why is this bad? Because these constants are unrelated to each other. This class is the only thing that they have in common. And the reference to this class will pollute many again unrelated components of the application. You want to later extract a component and use it in a different application? Or share some classes between a server and a remote client? You may need to ship the constants class as well! This class has introduced a dependency between otherwise unrelated components. This inhibits reuse and loose coupling and gives way to chaos.

Instead put constants where they belong. In no case should constants be used across component boundaries. This is only allowed if the component is a library, on which an explicit dependency is wanted.

Not noticing overflows

```
public int getFileSize(File f) {
  long l = f.length();
  return (int) l;
}
```

This developer, for whatever reason, wrapped a call to determine the size of a file into a method that returns an int instead of a long. This code does not support files larger than 2 GB and just returns a wrong length in that case. Code that casts a value to a smaller size type must first check for a possible overflow and throw an exception.

```
public int getFileSize(File f) {
  long l = f.length();
  if (l > Integer.MAX_VALUE) throw new IllegalStateException("int overflow");
  return (int) l;
}
```

Another version of an overflow bug is the following. Note the missing parantheses in the first println statement.

```
long a = System.currentTimeMillis();
long b = a + 100;
System.out.println((int) b-a);
System.out.println((int) (b-a));
```

And last, a true gem that I uprooted during code review. Note how the programmer tried to be careful, but then failed so badly by assuming an int could ever become larger than its maximum value.

```
int a = l.size();
a = a + 100;
if (a > Integer.MAX_VALUE)
    throw new ArithmeticException("int overflow");
```

Using == with float or double

```
for (float f = 10f; f!=0; f-=0.1) {
  System.out.println(f);
}
```

The above code doesn't behave as expected. It causes an endless loop. Because 0.1 is an infinite binary decimal, f will never be exactly 0. Generally you should never compare float or double values with the equality operator ==. Always use less than or greater than. Java compilers should be changed to issue a warning in that case. Or even make == an illegal operation for floating point types in the Java Language Spec. It makes really no sense to have this feature.

```
for (float f = 10f; f>0; f-=0.1) {
  System.out.println(f);
}
```

Storing money in floating point variables

```
float total = 0.0f;
for (OrderLine line : lines) {
  total += line.price * line.count;
}
double a = 1.14 * 75; // 85.5 represented as 85.4999...
System.out.println(Math.round(a)); // surprising output: 85
System.out.println(10.0/3); // surprising output: 3.3333333333333335 (precision lost twice during division and on conversion to decimal)
BigDecimal d = new BigDecimal(1.14); // precision has already been lost
```

I have seen many developers coding such a loop. Including myself in my early days. When this code sums 100 order lines with every line having one 0.30$ item, the resulting total is calculated to exactly 29.999971. The developer notices the strange behaviour and changes the float to the more precise double, only to get the result 30.000001192092896. The somewhat surprising result is of course due to the difference in representation of numbers by humans (in decimal format) and computers (in binary format). It always occurs in its most annyoing form when you add fractional amounts of money or calculate the VAT.

Binary representation of floating point numbers was invented for inherently *inexact* values like measurements. Perfect for engineering! But unusable when you want exact math. Like banks. Or when counting.

There are business cases where you can not afford to lose precision. You lose precision when converting between decimal and binary and when rounding happens in not a well-defined mannor or at indeterminate points. To avoid losing precision you must use fixed point or integer arithmetics. This does not only apply to monetary values, but it is a frequent source of annoyance in business applications and therefore makes a good example. In the second example an unsuspecting user of the program would simply say the computer's calculator is broken. That's of course very embarassing for the programmer.

Consequently an amount of money should *never ever* be stored in a floating point data type (float, double). Please note that it is not just any calculation that is inexact. Even a simple multiplication with an integer can already yield an inexact result. It is the mere fact of *storing* a value in a binary representation (float, double) that may already cause rounding! *You simply can not store 0.3 as an exact value in float or double*. Because float and double are binary IEEE754 types. If you see a float or double in your financial code base, the code will most likely yield inexact results. Instead either a string or fixed point representation should be chosen. A text representation must be in a well-defined format and is *not* to be confused with user input/output in a locale specific format. Both representations must define the precision (number of digits before and after the decimal point) that is stored.

For calculations the class BigDecimal provides an excellent facility. The class can be used such that it throws runtime exceptions if precision is unexpectedly lost in an operation. This is very helpful to uproot subtle numerical bugs and enables the developer to correct the calculation.

```
BigDecimal total = BigDecimal.ZERO;
for (OrderLine line : lines) {
  BigDecimal price = new BigDecimal(line.price);
  BigDecimal count = new BigDecimal(line.count);
  total = total.add(price.multiply(count)); // BigDecimal is immutable!
}
total = total.setScale(2, RoundingMode.HALF_UP);
```

```
BigDecimal a = (new BigDecimal("1.14")).multiply(new BigDecimal(75)); // 85.5 exact
a = a.setScale(0, RoundingMode.HALF_UP); // 86
System.out.println(a); // correct output: 86
BigDecimal a = new BigDecimal("1.14");
```

Not freeing resources in a finally block

```
public void save(File f) throws IOException {
  OutputStream out = new BufferedOutputStream(new FileOutputStream(f));
  out.write(...);
  out.close();
}

public void load(File f) throws IOException {
  InputStream in = new BufferedInputStream(new FileInputStream(f));
  in.read(...);
  in.close();
}
```

The above code opens an output stream to a file, allocating a file handle in the operating system. File handles are a rare resource and need to be properly freed, by calling close on the FileOutputStream (same for FileInputStream of course). To ensure that even in the case of an exception (the filesystem may become full during the write), closing must happen in a finally block. Here the stream is also wrapped into a buffering stream. That means not all data will have been written to disk by the time we arrive at the close() call. The close call itself will flush the pending data in the buffer to disk and may thus itself fail with an IOException. If that close fails the file on disk is incomplete (truncated) and thus probably corrupt. The method should therefore propagate the IOException in that case. In the case of a FileInputStream we can safely ignore the potential IOException from a close() call. We have read all data that we need, and there is nothing useful that we can do if the underlying close() failed anyway. It's not even worth logging it.

In a perfect world BufferedOutputStream.close() would be implemented correctly. But sadly it has a bug that's not going to be fixed: it loses any IOException from the implicit flush and truncates your file silently. So here we give the proper workaround with an explicit flush before close.

To be exact the corrected code below can leak in one small corner case: when the file stream was allocated but then allocating the buffered stream fails mysteriously (with out of memory for instance). As a pragmatic person I think in such a pathological case we can safely rely on the garbage collector to clean up the mess. It's not worth the hassle to deal with it.

```
// code for your cookbook
public void save() throws IOException {
  File f = ...
  OutputStream out = new BufferedOutputStream(new FileOutputStream(f));
  try {
    out.write(...);
    out.flush(); // don't lose exception by implicit flush on close
  } finally {
    out.close();
  }
}

public void load(File f) throws IOException {
  InputStream in = new BufferedInputStream(new FileInputStream(f));
  try {
    in.read(...);
  } finally {
    try { in.close(); } catch (IOException e) { }
  }
}
```

Let me give you also the cook book recipe for another ubiquitous pattern: database access. Again this is the pragmatic approach. Yes, rs.close() could fail with mysterious Errors, except they only occur in your university lecture on Quantum Mechanics and not in The Real World (tm). And only perverts would write the try/finally cascade that no Error neutrino can escape. Forgive my sarcasm. Here once and for all this is how to deal with SQL objects:

```
Car getCar(DataSource ds, String plate) throws SQLException {
  Car car = null;
  Connection c = null;
  PreparedStatement s = null;
  ResultSet rs = null;
  try {
    c = ds.getConnection();
    s = c.prepareStatement("select make, color from cars where plate=?");
    s.setString(1, plate);
    rs = s.executeQuery();
    if (rs.next()) {
       car = new Car();
       car.make = rs.getString(1);
       car.color = rs.getString(2);
    }
  } finally {
    if (rs != null) try { rs.close(); } catch (SQLException e) { }
    if (s != null) try { s.close(); } catch (SQLException e) { }
    if (c != null) try { c.close(); } catch (SQLException e) { }
  }
  return car;
}
```

With that said, don't miss the next paragraph.

Abusing finalize()

```
public class FileBackedCache {
   private File backingStore;

   ...

   protected void finalize() throws IOException {
      if (backingStore != null) {
        backingStore.close();
        backingStore = null;
      }
   }
}
```

This class uses the finalize method to release a file handle. The problem is that you can don't know when the method is called. The method is called by the garbage collector. If you are running out of file handles you want this method to be called rather sooner than later. But the GC will probably only invoke the method when you are about to run out of heap, which is a very different situation. It may take anything from milliseconds to days until GC and finalization runs. The garbage collector manages memory only. It does that very well. But it must not be abused to manage any other resources apart from that. **The GC is not a generic resource management mechanism!** I find Sun's API Doc of the finalize method very misleading in that respect. It actually suggest to use this method to close I/O resources - complete bullshit if you ask me. Again: I/O has *nothing* to do with memory!

Better code provides a public close method, which must be called by a well-defined lifecycle management, like JBoss MBeans or so.

```
public class FileBackedCache {
   private File backingStore;

   ...

   public void close() throws IOException {
      if (backingStore != null) {
        backingStore.close();
        backingStore = null;
      }
   }
}
```
JDK 1.7 (Java 7) will introduce the AutoClosable interface. It enables an automatic call to a close method, when the variable (not the object) goes out of scope of a try-with-resource block. It is very different from a finalizer. Its time of execution is well-defined at compile time.

```
try (Writer w = new FileWriter(f)) { // implements Closable
  w.write("abc");
  // w goes out of scope here: w.close() is called automatically in ANY case
} catch (IOException e) {
  throw new RuntimeException(e.getMessage(), e);
}
```

Involuntarily resetting Thread.interrupted

```
try {
		Thread.sleep(1000);
} catch (InterruptedException e) {
		// ok
}
```

or

```
while (true) {
		if (Thread.interrupted()) break;
}
```

The above code resets the interrupted flag of the Thread. Subsequent readers will not know that the Thread has been interrupted. If you need to pass on the information about the interrupt, rewrite the code like so.

```
try {
		Thread.sleep(1000);
} catch (InterruptedException e) {
		Thread.currentThread().interrupt();
}
```

or

```
while (true) {
            if (Thread.currentThread().isInterrupted()) break;
}
```

Spawning thread from static initializers

```
class Cache {
            private static final Timer evictor = new Timer();
}
```

java.util.Timer spwans a new thread in its constructor. Therefore the above code spawns a new thread in its static initializer. The new Thread will inherit some properties from its parent: context classloader, inheritable ThreadLocals, and some security properties (access rights). It is therefore rarely desireable to have those property set in an uncontrolled way. This may for instance prevent GC of a class loader.

The static initializer is executed by the thread that first loads the class (in any given ClassLoader), which may be a totally random thread from a thread pool of a webserver for example. If you want to control these thread properties you will have to start threads in a static method, and take control of who is calling that method.

```
class Cache {
   private static Timer evictor;

            public static setupEvictor() {
                        evictor = new Timer();
            }
}
```

Canceled timer tasks that keep state

```
final MyClass callback = this;
TimerTask task = new TimerTask() {
            public void run() {
                        callback.timeout();
            }
};
timer.schedule(task, 300000L);

try {
            doSomething();
} finally {
            task.cancel();
}
```

The above code uses a timer to enforce a timeout on doSomething(). The TimerTask contains an (implicit) instance reference to the outer class. Thus as long as the TimerTask exists the instance of MyClass may not be GC'ed. Unfortunately the Timer may keep cancelled TimerTasks around until their scheduled timeout has expired! That would leave the program 5 minutes with a dangling reference to the MyClass instance during which it can not get collected!

It's a temorary memory leak. A better TimerTask would override the cancel() method and null the reference there. It requires slightly more code.

```
TimerTask task = new Job(this);
timer.schedule(task, 300000L);

try {
        doSomething();
} finally {
        task.cancel();
}


static class Job extends TimerTask {
        private volatile MyClass callback;

        public Job(MyClass callback) {
                this.callback = callback;
        }

        public boolean cancel() {
                callback = null;
                return super.cancel();
        }

        public void run() {
                MyClass cb = callback;
                if (cb == null) return;
                cb.timeout();
        }
}
```

Holding strong references to ClassLoaders and unflushable caches

In a dynamic system like an application server or OSGI, you should take good care not to prevent ClassLoaders from garbage collection. As you undeploy and redeploy individual applications in an application server you create new class loaders for them. The old ones are unused and should be collected. Java isn't going to let that happen if there is a single dangling reference from container code into your application code.

As various libraries are used throughout an enterprise application, that directly means that libraries should do their very best not to hold involuntary strong references to objects (and thus their class loaders).

This is not easy. Classes like java.beans.Introspector from the JDK or org.apache.commons.beanutils.PropertyUtils from Apache BeanUtils or org.springframework.beans.CachedIntrospectionResults from Spring implement caches to speed up their inner workings. They keep strong references to classes you pass them for analysis. Fortunately they provide methods to flush their caches. But finding all classes that may have internal caches and flushing them at the right time is a near to impossible job for the developer.

If you happen to use org.apache.commons.el.BeanInfoManager from Apache Commons EL you probably have a leak. This ancient class keeps a cache of strong references that only ever grows until out of memory. And it has no flush method. Even Tomcat had to implement a workaround involving reflection to clean it.

It would be much better if these libraries just used soft or weak references in the first place. A quick reminder:

Soft and weak references basically differ in the point in time when they are nulled.

- WeakReference: nulled more or less at the same time when the last strong reference to the object goes away. Typical for classloader references (of what use is a classloader if none of its classes are loaded). But be careful if you use this *within* a ClassLoader implementation.
- SoftReference: the reference is kept even if the last strong reference to the object goes away as long as memory allows. Typical for caches.

Only if the library just caches objects from its own packages (with no external references), it may be fine not to use these special references and just use normal references.

Using soft or weak references also helps the runtime behaviour of your application: if memory gets tight, the last thing you want to spend memory on is caches. So the garbage collector will reclaim the memory used by caches if necessary. A bad example here is JBoss' SQL statement cache: it's compeletely static and can use a lot of memory, even when that is tight. Another bad example is JBoss' authentication cache.

Also every static cache must always provide a simple way to flush its contents. It's the nature of a (clean)cache (as opposed to e.g a write cache) that its contents are not valuable and can be safely discarded at any time. The limits of the cache are another trap. Caches should never grow large, and never cache objects for too long. A really bad example here is the default settings for the JDK DNS cache (it completely ignores DNS record lifetimes, and stores negative lookups forever in an unbounded list). Your API documentation should state if and when caching happens. This also helps the user to estimate runtime performance.

Anexo C –Google Java Style

Extraído de https://google-styleguide.googlecode.com/svn/trunk/javaguide.html (12/11/2014)

Google Java Style

Last changed: March 21, 2014

1 Introduction

This document serves as the **complete** definition of Google's coding standards for source code in the Java™ Programming Language. A Java source file is described as being *in Google Style* if and only if it adheres to the rules herein.

Like other programming style guides, the issues covered span not only aesthetic issues of formatting, but other types of conventions or coding standards as well. However, this document focuses primarily on the **hard-and-fast rules** that we follow universally, and avoids giving *advice* that isn't clearly enforceable (whether by human or tool).

1.1 Terminology notes

In this document, unless otherwise clarified:

1. The term *class* is used inclusively to mean an "ordinary" class, enum class, interface or annotation type (@interface).
2. The term *comment* always refers to *implementation* comments. We do not use the phrase "documentation comments", instead using the common term "Javadoc."

Other "terminology notes" will appear occasionally throughout the document.

1.2 Guide notes

Example code in this document is **non-normative**. That is, while the examples are in Google Style, they may not illustrate the *only* stylish way to represent the code. Optional formatting choices made in examples should not be enforced as rules.

2 Source file basics

2.1 File name

The source file name consists of the case-sensitive name of the top-level class it contains, plus the .java extension.

2.2 File encoding: UTF-8

Source files are encoded in **UTF-8**.

2.3 Special characters

2.3.1 Whitespace characters

Aside from the line terminator sequence, the **ASCII horizontal space character** (**0x20**) is the only whitespace character that appears anywhere in a source file. This implies that:

1. All other whitespace characters in string and character literals are escaped.
2. Tab characters are **not** used for indentation.

2.3.2 Special escape sequences

For any character that has a special escape sequence (\b, \t, \n, \f, \r, \", \' and \\), that sequence is used rather than the corresponding octal (e.g. \012) or Unicode (e.g. \u000a) escape.

2.3.3 Non-ASCII characters

For the remaining non-ASCII characters, either the actual Unicode character (e.g. ∞) or the equivalent Unicode escape (e.g. \u221e) is used, depending only on which makes the code **easier to read and understand**.

Tip: In the Unicode escape case, and occasionally even when actual Unicode characters are used, an explanatory comment can be very helpful.

Examples:

Example	**Discussion**
String unitAbbrev = "μs";	Best: perfectly clear even without a comment.
String unitAbbrev = "\u03bcs"; // "μs"	Allowed, but there's no reason to do this.
String unitAbbrev = "\u03bcs"; // Greek letter mu, "s"	Allowed, but awkward and prone to mistakes.
String unitAbbrev = "\u03bcs";	Poor: the reader has no idea what this is.
return '\ufeff' + content; // byte order mark	Good: use escapes for non-printable characters, and comment if necessary.

Tip: Never make your code less readable simply out of fear that some programs might not handle non-ASCII characters properly. If that should happen, those programs are **broken** and they must be **fixed**.

3 Source file structure

A source file consists of, **in order**:

1. License or copyright information, if present
2. Package statement
3. Import statements
4. Exactly one top-level class

Exactly one blank line separates each section that is present.

3.1 License or copyright information, if present

If license or copyright information belongs in a file, it belongs here.

3.2 Package statement

The package statement is **not line-wrapped**. The column limit (Section 4.4, Column limit: 80 or 100) does not apply to package statements.

3.3 Import statements

3.3.1 No wildcard imports

Wildcard imports, static or otherwise, **are not used**.

3.3.2 No line-wrapping

Import statements are **not line-wrapped**. The column limit (Section 4.4, Column limit: 80 or 100) does not apply to import statements.

3.3.3 Ordering and spacing

Import statements are divided into the following groups, in this order, with each group separated by a single blank line:

1. All static imports in a single group
2. com.google imports (only if this source file is in the com.google package space)
3. Third-party imports, one group per top-level package, in ASCII sort order
    - for example: android, com, junit, org, sun
4. java imports
5. javax imports

Within a group there are no blank lines, and the imported names appear in ASCII sort order. (**Note:** this is not the same as the import *statements* being in ASCII sort order; the presence of semicolons warps the result.)

3.4 Class declaration

3.4.1 Exactly one top-level class declaration

Each top-level class resides in a source file of its own.

3.4.2 Class member ordering

The ordering of the members of a class can have a great effect on learnability, but there is no single correct recipe for how to do it. Different classes may order their members differently.

What is important is that each class order its members in ***some* logical order**, which its maintainer could explain if asked. For example, new methods are not just habitually added to the end of the class, as that would yield "chronological by date added" ordering, which is not a logical ordering.

3.4.2.1 Overloads: never split

When a class has multiple constructors, or multiple methods with the same name, these appear sequentially, with no intervening members.

4 Formatting

Terminology Note: *block-like construct* refers to the body of a class, method or constructor. Note that, by Section 4.8.3.1 on array initializers, any array initializer *may* optionally be treated as if it were a block-like construct.

4.1 Braces

4.1.1 Braces are used where optional

Braces are used with if, else, for, do and while statements, even when the body is empty or contains only a single statement.

4.1.2 Nonempty blocks: K & R style

Braces follow the Kernighan and Ritchie style ("Egyptian brackets") for *nonempty* blocks and block-like constructs:

- No line break before the opening brace.
- Line break after the opening brace.
- Line break before the closing brace.
- Line break after the closing brace *if* that brace terminates a statement or the body of a method, constructor or *named* class. For example, there is *no* line break after the brace if it is followed by else or a comma.

Example:

```
return new MyClass() {
  @Override public void method() {
    if (condition()) {
      try {
        something();
      } catch (ProblemException e) {
        recover();
      }
    }
  }
};
```

A few exceptions for enum classes are given in Section 4.8.1, Enum classes.

4.1.3 Empty blocks: may be concise

An empty block or block-like construct *may* be closed immediately after it is opened, with no characters or line break in between ({}), **unless** it is part of a *multi-block statement* (one that directly contains multiple blocks: if/else-if/else or try/catch/finally).

Example:

```
  void doNothing() {}
```

4.2 Block indentation: +2 spaces

Each time a new block or block-like construct is opened, the indent increases by two spaces. When the block ends, the indent returns to the previous indent level. The indent level applies to both code and comments throughout the block. (See the example in Section 4.1.2, Nonempty blocks: K & R Style.)

4.3 One statement per line

Each statement is followed by a line-break.

4.4 Column limit: 80 or 100

Projects are free to choose a column limit of either 80 or 100 characters. Except as noted below, any line that would exceed this limit must be line-wrapped, as explained in Section 4.5, Line-wrapping.

Exceptions:

1. Lines where obeying the column limit is not possible (for example, a long URL in Javadoc, or a long JSNI method reference).
2. package and import statements (see Sections 3.2 Package statement and 3.3 Import statements).
3. Command lines in a comment that may be cut-and-pasted into a shell.

4.5 Line-wrapping

Terminology Note: When code that might otherwise legally occupy a single line is divided into multiple lines, typically to avoid overflowing the column limit, this activity is called *line-wrapping*.

There is no comprehensive, deterministic formula showing *exactly* how to line-wrap in every situation. Very often there are several valid ways to line-wrap the same piece of code.

Tip: Extracting a method or local variable may solve the problem without the need to line-wrap.

4.5.1 Where to break

The prime directive of line-wrapping is: prefer to break at a **higher syntactic level**. Also:

1. When a line is broken at a *non-assignment* operator the break comes *before* the symbol. (Note that this is not the same practice used in Google style for other languages, such as C++ and JavaScript.)
   - This also applies to the following "operator-like" symbols: the dot separator (.), the ampersand in type bounds (<T extends Foo & Bar>), and the pipe in catch blocks (catch (FooException | BarException e)).
2. When a line is broken at an *assignment* operator the break typically comes *after* the symbol, but either way is acceptable.
   - This also applies to the "assignment-operator-like" colon in an enhanced for ("foreach") statement.

3. A method or constructor name stays attached to the open parenthesis (() that follows it.
4. A comma (,) stays attached to the token that precedes it.

4.5.2 Indent continuation lines at least +4 spaces

When line-wrapping, each line after the first (each *continuation line*) is indented at least +4 from the original line.

When there are multiple continuation lines, indentation may be varied beyond +4 as desired. In general, two continuation lines use the same indentation level if and only if they begin with syntactically parallel elements.

Section 4.6.3 on Horizontal alignment addresses the discouraged practice of using a variable number of spaces to align certain tokens with previous lines.

4.6 Whitespace

4.6.1 Vertical Whitespace

A single blank line appears:

1. *Between* consecutive members (or initializers) of a class: fields, constructors, methods, nested classes, static initializers, instance initializers.
   - **Exception:** A blank line between two consecutive fields (having no other code between them) is optional. Such blank lines are used as needed to create *logical groupings* of fields.
2. Within method bodies, as needed to create *logical groupings* of statements.
3. *Optionally* before the first member or after the last member of the class (neither encouraged nor discouraged).
4. As required by other sections of this document (such as Section 3.3, Import statements).

Multiple consecutive blank lines are permitted, but never required (or encouraged).

4.6.2 Horizontal whitespace

Beyond where required by the language or other style rules, and apart from literals, comments and Javadoc, a single ASCII space also appears in the following places **only.**

1. Separating any reserved word, such as if, for or catch, from an open parenthesis (() that follows it on that line
2. Separating any reserved word, such as else or catch, from a closing curly brace (}) that precedes it on that line
3. Before any open curly brace ({), with two exceptions:
   - @SomeAnnotation({a, b}) (no space is used)
   - String[][] x = {{"foo"}}; (no space is required between {{, by item 8 below)
4. On both sides of any binary or ternary operator. This also applies to the following "operator-like" symbols:
   - the ampersand in a conjunctive type bound: <T extends Foo & Bar>
   - the pipe for a catch block that handles multiple exceptions: catch (FooException | BarException e)
   - the colon (:) in an enhanced for ("foreach") statement

5. After ,:; or the closing parenthesis ()) of a cast
6. On both sides of the double slash (//) that begins an end-of-line comment. Here, multiple spaces are allowed, but not required.
7. Between the type and variable of a declaration: List<String> list
8. *Optional* just inside both braces of an array initializer
   - new int[] {5, 6} and new int[] { 5, 6 } are both valid

Note: This rule never requires or forbids additional space at the start or end of a line, only *interior* space.

4.6.3 Horizontal alignment: never required

Terminology Note: *Horizontal alignment* is the practice of adding a variable number of additional spaces in your code with the goal of making certain tokens appear directly below certain other tokens on previous lines.

This practice is permitted, but is **never required** by Google Style. It is not even required to *maintain* horizontal alignment in places where it was already used.

Here is an example without alignment, then using alignment:

```
private int x; // this is fine
private Color color; // this too

private int   x;      // permitted, but future edits
private Color color;  // may leave it unaligned
```

Tip: Alignment can aid readability, but it creates problems for future maintenance. Consider a future change that needs to touch just one line. This change may leave the formerly-pleasing formatting mangled, and that is **allowed**. More often it prompts the coder (perhaps you) to adjust whitespace on nearby lines as well, possibly triggering a cascading series of reformattings. That one-line change now has a "blast radius." This can at worst result in pointless busywork, but at best it still corrupts version history information, slows down reviewers and exacerbates merge conflicts.

4.7 Grouping parentheses: recommended

Optional grouping parentheses are omitted only when author and reviewer agree that there is no reasonable chance the code will be misinterpreted without them, nor would they have made the code easier to read. It is *not* reasonable to assume that every reader has the entire Java operator precedence table memorized.

4.8 Specific constructs

4.8.1 Enum classes

After each comma that follows an enum constant, a line-break is optional.

An enum class with no methods and no documentation on its constants may optionally be formatted as if it were an array initializer (see Section 4.8.3.1 on array initializers).

private enum Suit { CLUBS, HEARTS, SPADES, DIAMONDS }

Since enum classes *are classes*, all other rules for formatting classes apply.

4.8.2 Variable declarations

4.8.2.1 One variable per declaration

Every variable declaration (field or local) declares only one variable: declarations such as int a, b; are not used.

4.8.2.2 Declared when needed, initialized as soon as possible

Local variables are **not** habitually declared at the start of their containing block or block-like construct. Instead, local variables are declared close to the point they are first used (within reason), to minimize their scope. Local variable declarations typically have initializers, or are initialized immediately after declaration.

4.8.3 Arrays

4.8.3.1 Array initializers: can be "block-like"

Any array initializer may *optionally* be formatted as if it were a "block-like construct." For example, the following are all valid (**not** an exhaustive list):

```
new int[] {          new int[] {
  0, 1, 2, 3           0,
}                    1,
                     2,
new int[] {            3,
  0, 1,              }
  2, 3
}                  new int[]
                     {0, 1, 2, 3}
```

4.8.3.2 No C-style array declarations

The square brackets form a part of the *type*, not the variable: String[] args, not String args[].

4.8.4 Switch statements

Terminology Note: Inside the braces of a *switch block* are one or more *statement groups*. Each statement group consists of one or more *switch labels* (either case FOO: or default:), followed by one or more statements.

4.8.4.1 Indentation

As with any other block, the contents of a switch block are indented +2.

After a switch label, a newline appears, and the indentation level is increased +2, exactly as if a block were being opened. The following switch label returns to the previous indentation level, as if a block had been closed.

4.8.4.2 Fall-through: commented

Within a switch block, each statement group either terminates abruptly (with a break, continue, return or thrown exception), or is marked with a comment to indicate that execution will or *might* continue into the next statement group. Any comment that communicates the idea of fall-through is sufficient (typically // fall through). This special comment is not required in the last statement group of the switch block. Example:

```
switch (input) {
  case 1:
  case 2:
    prepareOneOrTwo();
    // fall through
  case 3:
    handleOneTwoOrThree();
    break;
  default:
    handleLargeNumber(input);
}
```

4.8.4.3 The default case is present

Each switch statement includes a default statement group, even if it contains no code.

4.8.5 Annotations

Annotations applying to a class, method or constructor appear immediately after the documentation block, and each annotation is listed on a line of its own (that is, one annotation per line). These line breaks do not constitute line-wrapping (Section 4.5, Line-wrapping), so the indentation level is not increased. Example:

```
@Override
@Nullable
public String getNameIfPresent() { ... }
```

Exception: A *single* parameterless annotation *may* instead appear together with the first line of the signature, for example:

```
@Override public int hashCode() { ... }
```

Annotations applying to a field also appear immediately after the documentation block, but in this case, *multiple* annotations (possibly parameterized) may be listed on the same line; for example:

```
@Partial @Mock DataLoader loader;
```

There are no specific rules for formatting parameter and local variable annotations.

4.8.6 Comments

4.8.6.1 Block comment style

Block comments are indented at the same level as the surrounding code. They may be in /* ... */ style or // ... style. For multi-line /* ... */ comments, subsequent lines must start with * aligned with the * on the previous line.

```
/*
 * This is         // And so          /* Or you can
 * okay.           // is this.       * even do this. */
 */
```

Comments are not enclosed in boxes drawn with asterisks or other characters.

Tip: When writing multi-line comments, use the /* ... */ style if you want automatic code formatters to re-wrap the lines when necessary (paragraph-style). Most formatters don't re-wrap lines in // ... style comment blocks.

4.8.7 Modifiers

Class and member modifiers, when present, appear in the order recommended by the Java Language Specification:

public protected private abstract static final transient volatile synchronized native strictfp

4.8.8 Numeric Literals

long-valued integer literals use an uppercase L suffix, never lowercase (to avoid confusion with the digit 1). For example, 3000000000L rather than 3000000000l.

5 Naming

5.1 Rules common to all identifiers

Identifiers use only ASCII letters and digits, and in two cases noted below, underscores. Thus each valid identifier name is matched by the regular expression \w+ .

In Google Style special prefixes or suffixes, like those seen in the examples name_, mName, s_name and kName, are **not** used.

5.2 Rules by identifier type

5.2.1 Package names

Package names are all lowercase, with consecutive words simply concatenated together (no underscores). For example, com.example.deepspace, not com.example.deepSpace or com.example.deep_space.

5.2.2 Class names

Class names are written in UpperCamelCase.

Class names are typically nouns or noun phrases. For example, Character or ImmutableList. Interface names may also be nouns or noun phrases (for example, List), but may sometimes be adjectives or adjective phrases instead (for example, Readable).

There are no specific rules or even well-established conventions for naming annotation types.

Test classes are named starting with the name of the class they are testing, and ending with Test. For example, HashTest or HashIntegrationTest.

5.2.3 Method names

Method names are written in lowerCamelCase.

Method names are typically verbs or verb phrases. For example, sendMessage or stop.

Underscores may appear in JUnit *test* method names to separate logical components of the name. One typical pattern is test<*MethodUnderTest*>_<*state*>, for example testPop_emptyStack. There is no One Correct Way to name test methods.

5.2.4 Constant names

Constant names use CONSTANT_CASE: all uppercase letters, with words separated by underscores. But what *is* a constant, exactly?

Every constant is a static final field, but not all static final fields are constants. Before choosing constant case, consider whether the field really *feels like* a constant. For example, if any of that instance's observable state can change, it is almost certainly not a constant. Merely *intending* to never mutate the object is generally not enough. Examples:

```
// Constants
static final int NUMBER = 5;
static final ImmutableList<String> NAMES = ImmutableList.of("Ed", "Ann");
static final Joiner COMMA_JOINER = Joiner.on(',');  // because Joiner is immutable
static final SomeMutableType[] EMPTY_ARRAY = {};
enum SomeEnum { ENUM_CONSTANT }

// Not constants
static String nonFinal = "non-final";
final String nonStatic = "non-static";
static final Set<String> mutableCollection = new HashSet<String>();
static final ImmutableSet<SomeMutableType> mutableElements = ImmutableSet.of(mutable);
static final Logger logger = Logger.getLogger(MyClass.getName());
static final String[] nonEmptyArray = {"these", "can", "change"};
```

These names are typically nouns or noun phrases.

5.2.5 Non-constant field names

Non-constant field names (static or otherwise) are written in lowerCamelCase.

These names are typically nouns or noun phrases. For example, computedValues or index.

5.2.6 Parameter names

Parameter names are written in lowerCamelCase.

One-character parameter names should be avoided.

5.2.7 Local variable names

Local variable names are written in lowerCamelCase, and can be abbreviated more liberally than other types of names.

However, one-character names should be avoided, except for temporary and looping variables.

Even when final and immutable, local variables are not considered to be constants, and should not be styled as constants.

5.2.8 Type variable names

Each type variable is named in one of two styles:

- A single capital letter, optionally followed by a single numeral (such as E, T, X, T2)
- A name in the form used for classes (see Section 5.2.2, Class names), followed by the capital letter T (examples: RequestT, FooBarT).

5.3 Camel case: defined

Sometimes there is more than one reasonable way to convert an English phrase into camel case, such as when acronyms or unusual constructs like "IPv6" or "iOS" are present. To improve predictability, Google Style specifies the following (nearly) deterministic scheme.

Beginning with the prose form of the name:

1. Convert the phrase to plain ASCII and remove any apostrophes. For example, "Müller's algorithm" might become "Muellers algorithm".
2. Divide this result into words, splitting on spaces and any remaining punctuation (typically hyphens).
   - *Recommended:* if any word already has a conventional camel-case appearance in common usage, split this into its constituent parts (e.g., "AdWords" becomes "ad words"). Note that a word such as "iOS" is not really in camel case *per se*; it defies *any* convention, so this recommendation does not apply.
3. Now lowercase *everything* (including acronyms), then uppercase only the first character of:
   - ... each word, to yield *upper camel case*, or

  - ... each word except the first, to yield *lower camel case*
4. Finally, join all the words into a single identifier.

Note that the casing of the original words is almost entirely disregarded. Examples:

Prose form	**Correct**	**Incorrect**
"XML HTTP request"	XmlHttpRequest	XMLHTTPRequest
"new customer ID"	newCustomerId	newCustomerID
"inner stopwatch"	innerStopwatch	innerStopWatch
"supports IPv6 on iOS?"	supportsIpv6OnIos	supportsIPv6OnIOS
"YouTube importer"	YouTubeImporter YoutubeImporter*	

*Acceptable, but not recommended.

Note: Some words are ambiguously hyphenated in the English language: for example "nonempty" and "non-empty" are both correct, so the method names checkNonempty and checkNonEmpty are likewise both correct.

6 Programming Practices

6.1 @Override: always used

A method is marked with the @Override annotation whenever it is legal. This includes a class method overriding a superclass method, a class method implementing an interface method, and an interface method respecifying a superinterface method.

Exception:@Override may be omitted when the parent method is @Deprecated.

6.2 Caught exceptions: not ignored

Except as noted below, it is very rarely correct to do nothing in response to a caught exception. (Typical responses are to log it, or if it is considered "impossible", rethrow it as an AssertionError.)

When it truly is appropriate to take no action whatsoever in a catch block, the reason this is justified is explained in a comment.

```
try {
  int i = Integer.parseInt(response);
  return handleNumericResponse(i);
} catch (NumberFormatException ok) {
  // it's not numeric; that's fine, just continue
}
return handleTextResponse(response);
```

Exception: In tests, a caught exception may be ignored without comment *if* it is named expected. The following is a very common idiom for ensuring that the method under test *does* throw an exception of the expected type, so a comment is unnecessary here.

```
try {
  emptyStack.pop();
  fail();
} catch (NoSuchElementException expected) {
}
```

6.3 Static members: qualified using class

When a reference to a static class member must be qualified, it is qualified with that class's name, not with a reference or expression of that class's type.

```
Foo aFoo = ...;
Foo.aStaticMethod(); // good
aFoo.aStaticMethod(); // bad
somethingThatYieldsAFoo().aStaticMethod(); // very bad
```

6.4 Finalizers: not used

It is **extremely rare** to override Object.finalize.

Tip: Don't do it. If you absolutely must, first read and understand *Effective Java* Item 7, "Avoid Finalizers," very carefully, and *then* don't do it.

7 Javadoc

7.1 Formatting

7.1.1 General form

The *basic* formatting of Javadoc blocks is as seen in this example:

```
/**
 * Multiple lines of Javadoc text are written here,
 * wrapped normally...
 */
public int method(String p1) { ... }
```

... or in this single-line example:

```
/** An especially short bit of Javadoc. */
```

The basic form is always acceptable. The single-line form may be substituted when there are no at-clauses present, and the entirety of the Javadoc block (including comment markers) can fit on a single line.

7.1.2 Paragraphs

One blank line—that is, a line containing only the aligned leading asterisk (*)—appears between paragraphs, and before the group of "at-clauses" if present. Each paragraph but the first has <p> immediately before the first word, with no space after.

7.1.3 At-clauses

Any of the standard "at-clauses" that are used appear in the order @param, @return, @throws, @deprecated, and these four types never appear with an empty description. When an at-clause doesn't fit on a single line, continuation lines are indented four (or more) spaces from the position of the @.

7.2 The summary fragment

The Javadoc for each class and member begins with a brief **summary fragment**. This fragment is very important: it is the only part of the text that appears in certain contexts such as class and method indexes.

This is a fragment—a noun phrase or verb phrase, not a complete sentence. It does **not** begin with A {@code Foo} is a..., or This method returns..., nor does it form a complete imperative sentence like Save the record.. However, the fragment is capitalized and punctuated as if it were a complete sentence.

Tip: A common mistake is to write simple Javadoc in the form /** @return the customer ID */. This is incorrect, and should be changed to /** Returns the customer ID. */.

7.3 Where Javadoc is used

At the *minimum*, Javadoc is present for every public class, and every public or protected member of such a class, with a few exceptions noted below.

Other classes and members still have Javadoc *as needed*. Whenever an implementation comment would be used to define the overall purpose or behavior of a class, method or field, that comment is written as Javadoc instead. (It's more uniform, and more tool-friendly.)

7.3.1 Exception: self-explanatory methods

Javadoc is optional for "simple, obvious" methods like getFoo, in cases where there *really and truly* is nothing else worthwhile to say but "Returns the foo".

Important: it is not appropriate to cite this exception to justify omitting relevant information that a typical reader might need to know. For example, for a method named getCanonicalName, don't omit its documentation (with the rationale that it would say only /** Returns the canonical name. */) if a typical reader may have no idea what the term "canonical name" means!

7.3.2 Exception: overrides

Javadoc is not always present on a method that overrides a supertype method.