**Item 1: Consider static factory methods instead of constructors**

**Example:**

/\*\*

\* **@author** Sudhakar Duraiswamy

\*

\*/

**public** **class** Foo {

//Returns a new Instance everytime

**public** **static** Foo createNewFoo(){

**return** **new** Foo();

}

**public** **static** Foo *singletonFoo* = **null**;

//Maintains only one instance and returns the same

**public** **static** Foo getSingletonFoo(){

**if**(*singletonFoo* == **null**){

*singletonFoo* = **new** Foo();

}

**return** *singletonFoo*;

}

}

Advatages:

1. Unlike Constructors , Well-named static methods are self-explainatory and provide clear info on the how/what type of objects were constructed and returned.
2. Unlike Constructors , static methods are not required to create a new instance everytime they are called.

Example:

**public** **static** Foo *singletonFoo* = **null**;

//Maintains only one instance and returns the same

**public** **static** Foo getSingletonFoo(){

**if**(*singletonFoo* == **null**){

*singletonFoo* = **new** Foo();

}

**return** *singletonFoo*;

}

1. Unlike Constructors , they can return any SubType of their return type. More so the return type can also be decided.

Example

/\*\*

\* **@author** Sudhakar Duraiswamy

\*

\*/

**public** **class** Animal {

//Parent Class

}

//Another child class of Animal

**class** Tiger **extends** Animal{

}

//Child class of Foo

**class** Lion **extends** Animal{

**public** Lion(){

//This can construct only Object of Type Bar.class

}

//Returns any sub types of Foo

**public** **static** Animal createNewLion(){

**return** **new** Lion();

}

//Returns any sub types of Foo

**public** **static** Animal createNewTiger(){

**return** **new** Tiger();

}

}

1. Even a non-public class object can be constructed and exposed

In the below example , you can see , for Classes outside the ‘reptiles’ package can only access the Snake object , only through the Reptile.class there by enforcing strict guidelines to construct an Object

/\*\*

\* **@author** Sudhakar Duraiswamy

\*

\*/

**package** reptiles;

**public** **class** Reptile {

//Through static methods even class outside this package can

//use it

**public** **static** Snake createSnake(){

**return** **new** Snake();

}

}

//Snake has package access restriction- i.e only classes within the package reptiles can access it

**class** Snake{

}

One can make it more stricter , by providing private constuctor in Snake class , there by blocking all classes from constructing any Snake Instance outside of its class

**class** Snake{

//This prevents other classes from creating new instances of Snake

**private** Snake(){

}

//This is the only method through which a Snake instance could be created

**public** **static** Snake createSnake(){

**return** **new** Snake();

}

}

1. The class of the object returned by the static method, need even have to exist when you write the code.

String className="reptiles.Reptile";

Class<?> clazz = Class.*forName*(className);

Constructor<?> cstr = clazz.getConstructor();

Object object = cstr.newInstance(**new** Object[] { });

System.*out*.println("Reptile Constructed "+object);

Here are few many places this is used in Java

* Java.util.Java Collections Api – has many static methods for creating objects of various implementation of collections Interfaces
* Static factory methods form the basis of Service provider Framework example (eg. JCE) including service interface, service access API, provider interface, provider registration API.

DisAdvantages:

1. Using factory method pattern , will discourage developers from providing neccesary access to the classes , which in future may turn out be a big incovenience.

Ex. A class without a public or protected constructor cannot be subclassed.

1. The usability of the static factory method pattern , will soley depend on the naming convention and supported documentation .

Some established naming conventions are

getInstance()

valueof()

**Item 2: Consider a builder when faced with many constructor parameters**

Builder pattern allows the Objects constructed to be immutable, as the classes need not expose get and set methods for the buider classes to modify its state (as they are declared static nested classes).

Builder pattern allows abstraction of steps of construction of objects , which enables different implementation of these steps and construct different flavours(representation) of Objects.

Builder Pattern should be considered over the telescoping Constructor pattern only if the Constructor Arguments are over 4, because

1. Does not provide flexibility in setting parameters , as you are forced to pass a value for each argument defined in the constructor.
2. This advantage can be overcome , in conjunction with JavaBeans , but this prevents the class from being immutable and extra effort to ensure thread safety.

Recipies:

1. Designed in such a way , to create an object , you must create the builder.
2. Builder patter should only be considered when argument invariants are involved and the count exceeds 4.
3. Methods that take a Builder instance would typically constrain the builder’s type parameter using a bounded wildcard type

Tree buildTree(Builder<? **extends** Node> nodeBuilder) { ... }

1. If the builder is to be passed around for constructing one or more Objects , its best to define the Interface type .This ensures loose coupling.
2. The builder setter methods can return builder instance itself , so that invocations can be chained.

**public** **final** **class** Vehicle {

**private** **final** String type;

**private** **final** **int** wheels;

**private** Vehicle(Builder builder) {

**this**.type = builder.type;

**this**.wheels = builder.wheels;

}

**public** **static** Builder carBuilder() {

**return** **new** Builder("car");

}

**public** **static** Builder truckBuilder() {

**return** **new** Builder("truck");

}

**public** **static** **class** Builder {

**private** **final** String type;

**private** **int** wheels;

**private** Builder(String type) {

**this**.type = type;

}

**public** Builder addWheels(**int** wheels){

**this**.wheels = wheels;

**return** **this**;

}

**public** Vehicle build() {

**return** **new** Vehicle(**this**);

}

}

**public** **static** **void** main(String[] args) {

Vehicle car = Vehicle.*carBuilder*().addWheels(4).build();

Vehicle truck = Vehicle.*truckBuilder*().addWheels(10).build();

}

}

**Item 3: Enforce the singleton property with a private constructor**

**Can be implemented in 3 ways:**

* **Using Public field & private constructor**

/\*\*

\* **@author** Sudhakar Duraiswamy

\*

\*/

**public** **class** Singleton {

**public** **static** **final** Singleton *INSTANCE* = **new** Singleton();

**private** Singleton(){}

}

* **Public Static Factory method & private constructor (with lazy loading)**

**Note:One has to beware of Threading issues – hence Double check looking is needed.**

**public** **class** Singleton {

**private** **static** Singleton *INSTANCE* ;

**private** Singleton(){}

**public** **static** Singleton getInstance(){

**if**(**null** == *INSTANCE*){

**synchronized** (Singleton.**class**) {

**if**(**null** == *INSTANCE*) //Doubly checked

*INSTANCE* = **new** Singleton();

}

}

**return** *INSTANCE*;

}

}

* **Public Static Factory method & private constructor (with class loading)**

**public** **class** Singleton {

**public** **static** **final** Singleton *INSTANCE* = **new** Singleton();

**private** Singleton(){}

**public** **static** Singleton getInstance(){

**return** *INSTANCE*;

}

}

* **Singleton in Enum**

**public** **enum** EnumSingleton {

*INSTANCE*;

}

Note: Enums are thread safe.

Disadvantages of first two approaches is that the when these Object implement Serializable Interface , they no longer remain Singleton because the readObject() returns a new instance everytime it is called.This can be avoided by overriding readResolve() method.

Example:

/\*\*

\* **@author** Sudhakar Duraiswamy

\*

\*/

**public** **class** Singleton **implements** Serializable{

**private** **static** Singleton *INSTANCE* ;

**private** **int** number;

**private** Singleton(){}

**public** **static** Singleton getInstance(){

**if**(**null** == *INSTANCE*){

**synchronized** (Singleton.**class**) {

**if**(**null** == *INSTANCE*){ //Doubly checked

*INSTANCE* = **new** Singleton();

*INSTANCE*.number =5;

}

}

}

**return** *INSTANCE*;

}

**private** **void** writeObject(java.io.ObjectOutputStream out)

**throws** IOException{

out.writeObject(*INSTANCE*);

}

**private** **void** readObject(java.io.ObjectInputStream in)

**throws** IOException, ClassNotFoundException{

Singleton singleton = (Singleton)in.readObject();

System.*out*.println(singleton);

}

**private** Object readResolve(){

**return** *INSTANCE*;

}

**public** **static** **void** main(String[] args) **throws** Exception{

Singleton singleton = Singleton.*getInstance*();

ObjectOutputStream os= **new** ObjectOutputStream(**new** FileOutputStream(**new** File("singleton.temp")));

os.writeObject(singleton);

//Reading

ObjectInputStream is = **new** ObjectInputStream(**new** FileInputStream(**new** File("singleton.temp")));

Singleton o = (Singleton)is.readObject();

is = **new** ObjectInputStream(**new** FileInputStream(**new** File("singleton.temp")));

Singleton o2 = (Singleton)is.readObject();

System.*out*.println("is objects equal : " + (o == o2) );

}

}

**Item 4 :Enforce noninstantiability with a private constructor**

Some utility classes in java like java.util.Math , java.util.Arrays are designed to just group set of helper methods

And they are made non-instantiable by providing a private constructor.

**Item 5: Avoid creating unnecessary objects**

Avoid creating new Objects , unnecessarily.

Example

String wrong = new String (“hi , how are you”); //This always creates a new instance.

String right = (“hi , how are you”);//This allows JVM to reuse Strings from the String pool.

All immutable objects , can be made instance variables , initialized once and reused

//Correct

**private** **static** **final** SimpleDateFormat *sdf* = **new** SimpleDateFormat("mm-dd-yyyy");

**private** String getTodaysDate(){

**return** *sdf*.format(**new** Date());

}

//wrong

**private** String getTodaysDate1(){

SimpleDateFormat sdfNew = **new** SimpleDateFormat("mm-dd-yyyy");

**return** sdfNew.format(**new** Date());

}

Use String Appending classes like StringBuilder to append strings

//Wrong

String s = "this is ";

s = s +" wrong";

//Right

String s1 = **new** StringBuilder().append("this is").append(" correct").toString();

Note:

Please beware that reusing of Objects only were applicable , as this could lead to insidious issues which are very difficult to trace especially in threaded environment.

**Item 6: Eliminate obsolete object references**

Java Garbage Collector will garbage collect and reclaim the memory from only those objects that are de-referenced.

Its best advisable to implicitly de-reference objects by explicitly setting null.

Some Common Leak Scenarios are Unintentional Retention in collections, Cache

Some alternative

* To use WeakHashMap when implementing custom caches.
* Explicitly set Object variables to null, if they are not going to be used.

**Item 7: Avoid finalizers**

Also Note that you can mandate to call finalize() method , hence it is strongly advisable to place any critical code that needs to executed,Hence avoid using finalizers.

Use it as a as a safety net or to terminate noncritical native resources.

Also the “finalizer-chaining” is not performed automatically, hence those classed which override finalize() method should call super.finalize() to ensure parent class finalize method is called.

//Finalize method in subclass

**protected** **void** finalize() **throws** Throwable {

**try** {

//

} **finally** {

**super**.finalize();//Manual Finalizer chaining

}

}

**Item 8: Obey the general contract when overriding equals**

Contract for overriding equals

Reflexive, Symmetric, Transitive, Consistent

Recipe for High Quality Equals:

1. Use the == operator to check if the argument is a reference to **this** object.If it is below steps can be skipped and returned true.
2. Use the instanceof operator to check if the argument is of the correct type.
3. Cast the argument to the correct type.
4. For each “significant” field in the class, check to see if that field of the argument

matches the corresponding field of this object

1. Check if the implementation is Reflexive, Symmetric, Transitive, Consistent
2. Equals method should only compare Objects of its same Object Type

**Item 9: Always override hashCode when you override equals**

Contract : Objects which are equal (as per equals method ) should generate same hashcode.

Recipe for Hashcode:

@Override

**public** **int** hashCode() {

**int** nonZeroNumber = 31;

**int** constant = 0;

/\*

\*

\*

boolean: (f?1:0).

byte, char, short, or int: (int) f

long: (int) (f ^ (f >>> 32))

float: Float.floatToIntBits(f)

double: Double.doubleToLongBits(f)

null: 0

\*

\*

\*/

**int** hashcodevalue = 17 \* nonZeroNumber + constant;

**return** hashcodevalue;

}

**Item 10: Always override toString**

**public** **class** Foo {

**private** String name;

**private** **int** age;

**public** **static** **void** main(String[] args) {

Foo foo = **new** Foo();

foo.name="Sudhakar";

foo.age=26;

System.*out*.println(foo);

}

}

**Prints**

Foo@3e25a5

Overriding toString()

**public** **class** Foo {

**private** String name;

**private** **int** age;

@Override

**public** String toString() {

StringBuffer sf = **new** StringBuffer();

sf.append("name="+name);

sf.append(",age="+age);

**return** sf.toString();

}

**public** **static** **void** main(String[] args) {

Foo foo = **new** Foo();

foo.name="Sudhakar";

foo.age=26;

System.*out*.println(foo);

}

}

Prints

name=Sudhakar,age=26

General Recipes:

1. To String output should be concise and informative.
2. The method’s output should include all of the interesting information contained.
3. Whether or not you decide to specify the format, you should clearly document

your intentions

1. Provide programmatic access to all of the information contained in the value returned by toString.

**Item 11: Override clone judiciously**

A class doesn't have to provide its own implementation of clone in order to be cloneable. It can delegate that to its cloneable superclass.

**public class MyClass implements Cloneable {**

**protected Object clone() {**

**MyClass cloned = (MyClass) super.clone();**

**// set additional clone properties here**

**}**

**}**

clone must always return an instance of the same class as the instance it is called on.

That is impossible to achieve in the described case if an explicit constructor is called.

If the class overridng clone is final, on the other hand, this would be fine.

Hence a non-final class has to constructed without using its constructor and problem arises when mutable objects are used.

**public class A implements Cloneable {**

**public Object clone() {**

**return new A();**

**}**

**}**

**public class B extends A {**

**public Object clone() {**

**B b = (B)super.clone(); // <== will throw ClassCastException**

**}**

**}**

General Recipe for providing clone implemenation

1. All classes that implement Clonnable should override clone with a public method and first should call super.clone() and then fix any fields that need fixing (example Deep Copying.)
2. A good clone method will do a field-to-field copy of the class.
3. If the class only contains primitives and references to immutable objects then default fuctionality Is sufficient , else deep copy is needed.

Alternative would be to provide copy constrctor or copy factory method.

public Foo (Foo fooToCopy); //copy constructor

public static Foo newInstance(Foo fooToCopy); // copy factory method

**Item 12: Consider implementing Comparable**

public interface Comparable<T> {

int compareTo(T t);

}

Implementing Comparable is preferred when sorting among same object types is needed.This enables the comparison logic to be localised to an Object type ( instead of scattered in Util methods)and abstracted (better readability and maintainability.)

Recipe:

1. The compareTo() should only allow comparison to its object type, else should throw ClassCastException
2. It should be in consistent with Object equal method ie Reflexive, transitive and consistent.
3. (x.compareTo(y)==0) == (x.equals(y))
4. Comparison should start with the most signficant down to least significant.
5. If the objects are equal , then compareTo should return 0.
6. TreeSet , TreeMap and utility classes like Collections use Comparable for Sorting.

**Item 13: Minimize the accessibility of classes and members**

Make Class and class members as inaccessible as possible. This ensures good encapsulation(information hiding).

If a method overrides a super class method, it is not allowed to have lower access level than that of the method in super class. Java Compiler ensures that this followed. This ensures the instance of the subclass can be used anywhere an instance of Super class can be used.

General Recipes:

1. Make each class and its members as inaccessible as possible which allows proper functioning.
2. Choose the least accessible **package-private** for Classes , unless need be don’t declare **public** classes
3. Avoid Inheritance unless it is really necessary. If a method overrides a super class method, it is not allowed to have lower access level than that of the method in super class. This goes against Encapsulation

**Item 14: In public classes, use accessor methods, not public fields**

**Item 15: Favor immutability**

1. Make the immutable class to be final.
2. Immutable objects are inherently thread-safe; they require no synchroni-zation.
3. Use companion class such as StringBuilder replacing String for performance.
4. Static factory is a good way for immutable class.
5. Classes should be immutable unless there’s a very good reason to make them mutable.
6. Make every field final unless there is a compelling reason to make it nonfinal.
7. Don’t provide a public initialization method separate from the constructor or static factory unless there is a compelling reason to do so.

**Item 16: Favor composition over inheritance**

Inheritance breaks encapsulation, as it tightly depends on the implementation of Super Class, for its proper function.

Change in the implementation details of the Super class will have impact. Its effective use depends on how well the behavior of the super class is documented (i.e behavior when a method is override, what methods to override to perform a certain operation, sequence of processing etc. )

The Considerable advantages one can see using Inheritance is code reusability (which can achieved through Composition) and dynamic Polymorphism.

Hence for most cases, composition is better than Inheritance.

Refer this SO: <http://stackoverflow.com/questions/49002/prefer-composition-over-inheritance>

**Item 17: Design and document for inheritance or else prohibit it**

The class must document the effects of overriding any method in detail.

This includes

1. Behavior when a method is override
2. What methods to override to perform a certain operation,
3. Sequence of processing
4. How the method is invoked (i.e though a thread or static initializers or constructors etc.)

General Guidelines

1. The class must document precisely the effects of overriding any method.
2. A class may have to provide hooks into its internal workings in the form of judiciously chosen protected methods
3. Constructors must not invoke overridable methods, directly or indirectly
4. Neither clone nor readObject may invoke an overridable method, directly or indirectly
5. Prohibit subclassing in classes that are not designed and documented to be safely subclassed using **final** keyword.

**Item 18: Prefer interfaces to abstract classes**

1. Existing classes can be easily retrofitted to implement a new interface.
2. Interfaces are ideal for defining mixins.
3. Interfaces allow the construction of nonhierarchical type frameworks.
4. Interfaces enable safe, powerful functionality enhancements
5. You can combine the virtues of interfaces and abstract classes by providing an abstract
6. skeletal implementation class to go with each nontrivial interface that you export.
7. Using abstract classes to define types that permit multiple implementations has one great advantage over using interfaces: It is far easier to evolve an abstract class than it is to evolve an interface.

**Item 19: Use interfaces only to define types**

Interfaces should only used to define type,that can be used to refer to instances of the class.

Enum and Utility class are more appropriate for defining constants.

The constant Interface pattern is a poor use of interfaces, This is used to avoid the need to qualify the constants used in the class.

**public** **interface** SportsConstants {

**public** **static** **final** String *CRICKETBALL*="basket\_ball";

**public** **static** **final** **int** *CRICKET\_PLAYER\_COUNT*=11;

}

If the constants are closely tied to the Interface , then it should be defined within the Interface.

**Item 20: Prefer class hierarchies to tagged classes**

Tagged classes are cluttered with tag fields, and switch statements, messing the encapsulation and  being prone to run-time errors.

Tagged classes are usually clluttered with tag fields (i.e flow identifiers) , they can quickly grow out of hand and unmaintainable.

/\*\*

\* **@author** Sudhakar Duraiswamy

\*

\*/

**public** **class** Shape {

**enum** ShapeType { *RECTANGLE*, *SQUARE* };

**private** **int** length;

**private** **int** width;

**final** ShapeType type;

**public** Shape(ShapeType type ,**int** side) **throws** Exception{

**if**(!type.equals(ShapeType.*SQUARE*)) **throw** **new** Exception("Not Supported");

**this**.length = side;

**this**.type = type;

}

**public** Shape(ShapeType type ,**int** length,**int** width) **throws** Exception{

**if**(!type.equals(ShapeType.*RECTANGLE*)) **throw** **new** Exception("Not Supported");

**this**.length = length;

**this**.width = width;

**this**.type = type;

}

**public** String getShapeName(){

**return** type.toString();

}

//The method does entirely two different things based on the

//shape type .This called Tagging.

**public** **int** Area(){

**int** area =0;

**if**(type == ShapeType.*SQUARE*){

area = length\*length;

}**else** **if**(type == ShapeType.*RECTANGLE*){

area = length\*width;

}

**return** area;

}

}

Tagged class should be abandoned and replaced by abstract classes .Any common general functions can be defined in the Abstract super class and is available for classes extending it.

**public** **abstract** **class** Shape {

**enum** ShapeType { *RECTANGLE*, *SQUARE* };

**public** Shape(ShapeType type){

**this**.type = type;

}

**private** ShapeType type;

//Common Function/ behavior for all child classes.

**public** String getShapeName(){

**return** type.toString();

}

//All Child classes should provide their own implementation

**public** **abstract** **int** area();

}

Child Class :Rectange

**public** **class** Rectangle **extends** Shape{

**private** **int** length;

**private** **int** width;

**public** Rectangle(ShapeType type, **int** length,**int** width){

**super**(type);

**this**.length = length;

**this**.width=width;

}

**public** **int** area(){

**int** area = length\*length;

**return** area;

}

}

This class inherits getShapName() from its Parent Classes.

Child Class :Square

**public** **class** Square **extends** Rectangle{

**public** Square(**int** side){

**super**(ShapeType.*SQUARE*, side, side);

}

}

This class inherits all of the methods i.e getShapName() and area() from its Parent Classes.

**Item 21: Use function objects to represent strategies**

Function Object(AKA Strategy Class) is basically an instance of the class main functionality is implemented in a single function.

Example in Java would be java.util.Comparator Interface.

List<String> list = Arrays.*asList*("6", "8", "21", "11", "5");

Comparator<String> numberComparator = **new** Comparator<String>() {

**public** **int** compare(String str1, String str2) {

**return** Integer.*valueOf*(str1).compareTo(Integer.*valueOf*(str2));

}

};

Collections.*sort*(list, numberComparator);

Heres an example on how to define Custom Function Objects and use them

/\*\*

\* **@author** Sudhakar Duraiswamy

\*

\*/

**public** **class** Person {

**private** **int** age;

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

//This defines the contract for the Strategy

**interface** AgeModifier{

**void** modifyAge(Person person ,**int** i);

}

**public** **static** **void** main(String[] args) {

Person p1 = **new** Person();

p1.age = 25;

Person p2 = **new** Person();

p2.age = 35;

List<Person> list = Arrays.*asList*( **new** Person[]{p1,p2});

//Using Anonymous Inner classes to provide concrete implementation the Stratgy

//This increase the age by person

*performAction*(list, **new** AgeModifier(){

**public** **void** modifyAge(Person person, **int** i) {

person.setAge(person.getAge() + i);

}}, 5);

//This decreases Age by 5

*performAction*(list, **new** AgeModifier(){

**public** **void** modifyAge(Person person, **int** i) {

person.setAge(person.getAge() - i);

}}, 5);

}

**public** **static** **void** performAction(List<Person> list , AgeModifier operation,**int** increment){

**for**(Person person :list){

operation.modifyAge(person, increment);

}

}

}

Recipes

1. As Java Does not have first-class functions, Function Objects are expressed by an Interface with a single method and with implementation typically a Anonymous inner class.
2. If the Inner Class is to be used repeatedly , eg. Within a loop , then it make sense to use final , static Singleton instance of the inner class to perform the operation.This class which represents the strategy of the operation is also called Concrete Strategy Class.
3. Concrete Strategy Classes should be stateless.

PENDING

**Item 22: Favor static member classes over nonstatic**

Four kinds of nested classes: static member classes, nonstatic member classes, anonymous classes, and local classes.

It is impossible to create an instance of a nonstatic member class without an enclosing instance.

Example:

**public** **class** Calculator {

**private** **final** **int** x ;

**private** **final** **int** y;

**private** Calculator(**int** x,**int** y){

**this**.x= x;

**this**.y=y;

}

**public** Operation getOperation(){

**return** **new** Operation();

}

**class** Operation{

**public** Operation(){

System.*out*.println("Operation constructed");

}

**public** **int** add(){

**return** x+y;

}

**public** **int** multiply(){

**return** x\*y;

}

}

**public** **static** **void** main(String[] args) {

Calculator c = **new** Calculator(10, 2);

System.*out*.println(c.getOperation().add());

}

}

Declare a member class as static if it does not require access to an enclosing instance.

Example :

Builder pattern

Three common uses of anonymous classes:

1. Function objects
2. Process objects, such as Runnable, Thread
3. static factory methods

General Recipes:

1. If a nested class needs to be visible outside of a single method or is too long to fit comfortably inside a method, use a member class
2. If each instance of the member class needs a reference to its enclosing instance, make it nonstatic; otherwise,make it static.
3. Assuming the class belongs inside a method, if you need to create instances from only one location and there is a preexisting type that characterizes the class, make it an anonymous class; otherwise, make it a local class

**Item 23: Don’t use raw types in new code**

* Always use Generics unless you have strong reason not to do so, as the compiler will not check the type of the parameters , hence loses type safety, which may lead to Run time Error (esp. ClassCastException).
* Generics like List<Object> can contain any arbitrary types but still with type safety
* .Unbounded Wild Card types like List<?> is least restrictive in the sense it can hold any type of element. But you cannot put any element into Collection<?> other than null. If these restrictions are unacceptable use **Generic Methods** or **Bounded wild Card types**. The Unbounded Wild Card types should be used when you don’t really care about the type .

//This method functionality is not dependent on Specific Generic Type , hence

//declared using unbounded generic type.

**static** **int** getListSize(List<?> list){

**return** list.size();

}

* Generic type information is only applicable at compiled time and is erased at runtime.
* Can use Class literals like String.class , List.class as Runtime-type Tokens .For more info refer this link :

<http://docs.oracle.com/javase/tutorial/extra/generics/literals.html>

|  |  |  |
| --- | --- | --- |
| **Term** | **Example** | **Item No.** |
| Parameterized type | List<String> | 23 |
| Actual Type Parameter | String | 23 |
| Generic Type | List<E> | 23,26 |
| Formal type parameter | E | 23 |
| Unbounded Wildcard type | List<?> | 23 |
| Raw Type | List | 23 |
| Bounded type parameter | <E extends Number> | 26 |
| Recursive Type bound | <T extends Comparable<T>> | 27 |
| Bounded Wild Card type | <? extends Number> | 28 |
| Generic Method | static <E> List<E> asList(E[] a) | 27 |
| Type Token | String.class | 29 |

**Item 24: Eliminate unchecked warnings**

1. Eliminate every unchecked warnings as you can by fixing it.

If you can’t eliminate the warning and you can prove the code is typesafe then eliminate the warnings , using

@SuppressWarnings(“unchecked”)

1. Always use SuppressWarnings Annotation on the narrowest possible scope.
2. Every time you use an @SuppressWarnings("unchecked") annotation,add a comment saying why it’s safe to do so.

Eliminate the unchecked warning, including:

* unchecked cast warnings
* unchecked method invocation warnings
* unchecked generic array creation warnings
* unchecked conversion warnings

**Item 25: Prefer lists to arrays**

Arrays are covariant and reifiable. Generics are invariant and erased in run-time.

Arrays provide runtime type safety but not compile-time type safety and vice versa for generics.

|  |  |
| --- | --- |
| 1  2  3 | // Fails at runtime!  Object[] objectArray = new Long[1];  objectArray[0] = "I don't fit in"; // Throws ArrayStoreException |
| 1  2  3 | // Won't compile!  List<Object> ol = new ArrayList<Long>(); // Incompatible types  ol.add("I don't fit in"); |

Generics enforce their type constraints only at compile time and discard (or erase) their element type information at runtime.

it is illegal to create an array of a generic type such as “new List<String>[]” and “new List<E>[]“.

Non-reifiable type is one whose runtime representation contains less information than its compile-time representa-tion.

Casts to arrays of non-reifiable types should be used only under special circumstances. A better solution is to use list.

**Item 26: Favor generic types**

Generics are safer. Suppress the warning when casting.

|  |  |
| --- | --- |
| 1  2  3  4 | @SuppressWarnings("unchecked")  public Stack() {  elements = (E[]) new Object[DEFAULT\_INITIAL\_CAPACITY];  } |

**Item 27: Favor generic methods**

Generic singleton factory:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | // Generic singleton factory pattern  private static UnaryFunction<Object> IDENTITY\_FUNCTION =  new UnaryFunction<Object>()  {  public Object apply(Object arg) { return arg; }  };  // IDENTITY\_FUNCTION is stateless and its type parameter is  // unbounded so it's safe to share one instance across all types.  @SuppressWarnings("unchecked")  public static <T> UnaryFunction<T> identityFunction() {  return (UnaryFunction<T>) IDENTITY\_FUNCTION;  } |

Recursive type bound is usually used for Comparable<T>.

|  |  |
| --- | --- |
| 1  2  3  4  5 | public interface Comparable<T>; {  int compareTo(T o);  }  // Using a recursive type bound to express mutual comparability  public static <T extends Comparable<T>> T max(List<T> list) {...} |

**Item 28: Use bounded wildcards to increase API flexibility**

Producer-extends, consumer-super(PECS)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // Wildcard type for parameter that serves as an E producer  public void pushAll(Iterable<? extends E> src) {  for (E e : src)  push(e);  }  // Wildcard type for parameter that serves as an E consumer  public void popAll(Collection<? super E> dst) {  while (!isEmpty())  dst.add(pop());  } |

Do not use wildcard types as return types. Rather than providing additional flexibility for your users, it would force them to use wildcard types in client code.

Always use Comparable<? super T> in preference to Comparable<T>, since the type may implements the Comparable interface as an subinterface.

If a type parameter appears only once in a method declaration, replace it with a wildcard.

Can’t put any value except null into a List<?>, which means List of some particular type.

**Item 29: Consider typesafe heterogeneous containers**

Place the type parameter on the key rather than the container to enable the container to include different types.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | // Typesafe heterogeneous container pattern - implementation  public class Favorites {  private Map<Class<?>, Object> favorites =  new HashMap<Class<?>, Object>();  public <T> void putFavorite(Class<T> type, T instance) {  if (type == null)  throw new NullPointerException("Type is null");  favorites.put(type, instance);  }  public <T> T getFavorite(Class<T> type) {  return type.cast(favorites.get(type));  }  } |

**Item 30: Use enums instead of int constants**

Java’s enum types are full-fledged classes.

Enums provide high-quality implementations of all the Object methods, Comparable and Serializable.

Enums can also add methods or fields.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // Enum type with constant-specific method implementations  public enum Operation {  PLUS { double apply(double x, double y){return x + y;} },  MINUS { double apply(double x, double y){return x - y;} },  TIMES { double apply(double x, double y){return x \* y;} },  DIVIDE { double apply(double x, double y){return x / y;} };  abstract double apply(double x, double y);  } |

ValueOf(String) method can translate a constant’s name into the constant itself.

Consider the strategy enum pattern if multiple enum constants share common behaviors.

**Item 31: Use instance fields instead of ordinals**

Avoid using ordinal() method of enum.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // Abuse of ordinal to derive an associated value - DON'T DO THIS  public enum Ensemble {  SOLO, DUET, TRIO, QUARTET, QUINTET,  SEXTET, SEPTET, OCTET, NONET, DECTET;  public int numberOfMusicians() { return ordinal() + 1; }  } |

**Item 32: Use EnumSet instead of bit fields**

Use general Set interface for input type.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // EnumSet - a modern replacement for bit fields  public class Text {  public enum Style { BOLD, ITALIC, UNDERLINE, STRIKETHROUGH }  // Any Set could be passed in, but EnumSet is clearly best  public void applyStyles(Set<Style> styles) { ... }  } |

**Item 33: Use EnumMap instead of ordinal indexing**

It is better to use EnumMap to catalog relationship related with Enum type.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // Using an EnumMap to associate data with an enum  Map<Herb.Type, Set<Herb>> herbsByType =  new EnumMap<Herb.Type, Set<Herb>>(Herb.Type.class);  for (Herb.Type t : Herb.Type.values())  herbsByType.put(t, new HashSet<Herb>());  for (Herb h : garden)  herbsByType.get(h.type).add(h);  System.out.println(herbsByType); |

**Item 34: Emulate extensible enums with interfaces**

In general, extending enums is a bad idea. If you really want to do that, define an interface for expected behaviors.

**Item 35: Prefer annotations to naming patterns**

Annotations do not directly affect program semantics, but they do affect the way programs are treated by tools and libraries, which can in turn affect the semantics of the running program. Annotations can be read from source files, class files, or reflectively at run time.

Annotations complement javadoc tags. In general, if the markup is intended to affect or produce documentation, it should probably be a javadoc tag; otherwise, it should be an annotation.

**Item 36: Consistently use the Override annotation**

Use @Override annotation except for the abstract methods or methods of interfaces.

**Item 37: Use marker interfaces to define types**

Marker interface contains no method declarations, which indicates some property or features. The typical marker interface is Serialize interface.

**Item 38: Check parameters for validity**

Check and restrict the parameters of functions.

* Throw Exception
* Assert

**Item 39: Make defensive copies when needed**

In general, if a class has mutable components that it gets from or returns to its clients, the class must defensively copy these components.

Defensive copy usually implemented by:

* clone() method
* copy constructor
* static factory

This is not mandatory. If the class trusts the clients, then defensive copy could be replaced by documentation explanation to save the cost.

**Item 40: Design method signatures carefully**

Avoid long parameter lists.  Divide them to several methods or use helper method and helper class.

Favor interfaces over classes.

Prefer two-element enum types to boolean parameters for clearness and extendability.

**Item 41: Use overloading judiciously**

Selection among overloaded methods is static (at the compile time), while selection among overridden methods is dynamic.

Refrain using of overloading methods with same number of parameters.

If you can not avoid that, try avoiding parameters that can be passed to different overloadings by casts.

**Item 42: Use varargs judiciously**

Do not use varargs unless you really need it and benefit from it.

Varargs is expensive in resources.

Arrays.asList() is tricky. It returns List<Integer> when the argument is Integer[] and returns List<int[]> when the argument is int[].

**Item 43: Return empty arrays or collections, not nulls**

Returning null requires extra judgement in the client side.

Use new Object1[0] for arrays and emptyList(), emptySet() and emptyMap() for containers.

**Item 44: Write doc comments for all exposed API elements**

@param tag for the parameters. @return for return values. @throws for exception may throw.

Care the HTML metacharacters in doc comments.

Write succinct and nice comment as the summary description in the beginning.

For methods and constructors: verb phrase .

* Returns the number of elements in this collection.

For classes, interfaces, and fields: noun phrase.

* A task that can be scheduled for one-time or repeated execution by a Timer.

Document generic types and constants.

**Item 45: Minimize the scope of local variables**

Minimize the scope of a local variable by declaring it where it is first used.

Prefer  for loops to  while loops.

**Item 46: Prefer for-each loops to traditional for loops**

Use the for each loops to iterate the arrays.

Exception:

1. you need to remove the elements.
2. you need to change the value of elements.
3. special control for the pace of iteration.

**Item 47: Know and use the libraries**

Be familiar with the contents of java.lang, java.util, and, to a lesser extent, java.io.

**Item 48: Avoid  float and double  if exact answers are required**

Use the BigDecimal or int or long instead.

int for number with less than 9 digits and long for 19 digits.

**Item 49: Prefer primitive types to boxed primitives**

“==” operation to mixture of primitive and boxed primitive compares the reference identity but not value.

Use autoboxing carefully.

**Item 50: Avoid strings where other types are more appropriate**

Do not use string type in everywhere.

**Item 51: Beware the performance of string concatenation**

Use append() of StringBuilder instead.

**Item 52: Refer to objects by their interfaces**

Using interface as parameter type make the program more flexible.

List<Subscriber> subscribers = new Vector<Subscriber>();

List<Subscriber> subscribers = new ArrayList<Subscriber>();

**Item 53: Prefer interfaces to reflection**

Reflection loses all information of complile-time check.

Reflection is only used at design time.

When using the class unknown at compile time, try to only instantiate them by reflection. Access fields and call methods using interfaces or superclasses.

**Item 54: Use native methods judiciously**

Avoid using native code unless you really need it.

**Item 55: Optimize judiciously**

Weshould  forget about small efficiencies, say about 97% of the time: prema-ture optimization is the root of all evil.  –Donald E. Knuth.

Write good programs but not fast programs.

**Item 56: Adhere to generally accepted naming conventions**

Generally, follow The Java Language Specification as conventions.

*Package*    com.google.inject, org.joda.time.format  
*Class or Interface*     Timer, FutureTask, LinkedHashMap, HttpServlet  
*Method or Field*    remove, ensureCapacity, getCrc  
*Constant Field*      MIN\_VALUE, NEGATIVE\_INFINITY  
*Local Variable*    i, xref, houseNumber  
*Type Parameter*    T, E, K, V, X, T1, T2

**Item 57: Use exceptions only for exceptional conditions**

Do not use the exception for control flow, exception is designed for exceptional condition.

**Item 58: Use checked exceptions for recoverable conditions and runtime exceptions for programming errors**

Checked exception is used in recoverable case. For that, it need to provide information or methods to help the caller to recover.

**Item 59: Avoid unnecessary use of checked exceptions**

Avoid using the checked exception unless:

* the exception can not avoided by proper use of API.
* Handling exception brings benefits.

**Item 60: Favor the use of standard exceptions**

Some commonly used exceptions:

IllegalArgumentException: Non-null parameter value is inappropriate  
IllegalStateException: Object state is inappropriate for method invocation  
NullPointerException: Parameter value is null where prohibited  
IndexOutOfBoundsException: Index parameter value is out of range  
ConcurrentModificationException: Concurrent modification of an object has been detected where it is prohibited  
UnsupportedOperationException: Object does not support method

**Item 61: Throw exceptions appropriate to the abstraction**

If it is impossible to prevent the occurring of exception in lower level, propagate it to higher level by catching and rethrowing.

Try to log the exception information if detailed information is needed.

**Item 62: Document all exceptions thrown by each method**

Document the exceptions by Javadoc @throws tag. Do not “throws” unchecked exceptions.

**Item 63: Include failure-capture information in detail messages**

Include failure-capture information in checked exceptions. Those information is useful in recovering from failure.

**Item 64: Strive for failure atomicity**

Any generated exception should leave the object in the same state it was in prior to the method invocation.

Example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public Object pop() {  if (size == 0)  throw new EmptyStackException();  Object result = elements[--size];  elements[size] = null; // Eliminate obsolete reference  return result;  } |

Do not try to maintain the failure atomicity when throwing errors.

**Item 65: Don’t ignore exceptions**

Do not use empty catch block to ignore the exception. Throw it outward at least if you do not know how to handle it.

**Item 66: Synchronize access to shared mutable data**

Synchronization includes two parts: mutual exclusion and visibility. Without synchronization, one thread’s changes might not be visible to other threads.

Do not use  Thread.stop as it is depreciated.

Synchronization has no effect unless both read and write operations are synchronized.

Volatile and Atomic types should be carefully used.

**Item 67: Avoid excessive synchronization**

Do not call alien methods in synchronized block. Doing this is uncontrollable and may cause exceptions and deadlock.

Use concurrent containers like CopyOnWriteArrayList to separate the data writing and reading. This is particularly useful to the situation in which writing data is happened occasionally.

In a multicore world, the real cost of excessive synchronization is not the CPU time spent obtaining locks; it is the lost opportunities for parallelism and the delays imposed by the need to ensure that every core has a consistent view of memory.

Do not synchronize the class internally unless you have good reason.

**Item 68: Prefer executors and tasks to threads**

Executor framework separate the task and mechanism of executing. So use executors prior to  Thread.

Using Executors.newFixedThreadPool under heavy loading situations.

Using ScheduledThreadPoolExecutor prior to Timer when multiple timing tasked are required.

**Item 68: Prefer executors and tasks to threads**

The higher-level utilities in java.util.concurrent fall into three categories:the Executor Framework, concurrent collections; and synchronizers.

Utilities in the concurrent library should be considered first before wait() and notify().

Use  ConcurrentHashMap in preference to  Collections.synchronizedMap  or  Hashtable.

Common synchronizers include CyclicBarrier, CountdownLatch and Semaphore.

**Item 70: Document thread safety**

The presence of the synchronized modifierin a method declaration is an implementation detail, not a part of its exported API.

To enable safe concurrent use, a class must clearly document what level of thread safety it supports.

levels of thread safety:

* immutable
* unconditionally thread-safe
* conditionally thread-safe
* not thread-safe

**Item 71: Use lazy initialization judiciously**

Lazy initialization decreases the cost of initializing a class or creating an instance, at the expense of increasing the cost of accessing the lazily initialized field. It may be worthwhile when only part of instances will be initialized in practice.

Use a synchronized accessor to the getfield method to ensure the concurrency.

Use the lazy initialization holder class idiom for static field.

|  |  |
| --- | --- |
| 1  2  3  4  5 | // Lazy initialization holder class idiom for static fields  private static class FieldHolder {  static final FieldType field = computeFieldValue();  }  static FieldType getField() { return FieldHolder.field; } |

Use the double-check idiom for instance field.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | // Double-check idiom for lazy initialization of instance fields  private volatile FieldType field;  FieldType getField() {  FieldType result = field;  if (result == null) { // First check (no locking)  synchronized(this) {  result = field;  if (result == null) // Second check (with locking)  field = result = computeFieldValue();  }  }  return result;  } |

**Item 72: Don’t depend on the thread scheduler**

Do not rely on scheduler for correctness of program.

Do not let the thread busy-wait.

Use Thread.sleep(1) instead of Thread.yield() for increasing concurrency.

**Item 73: Avoid thread groups**

Thread groups are obsolete.

**Item 74: Implement  Serializable judiciously**

Long term cost of implementing Serializable include:

Decreases the flexi-bility to change a class’s implementation once it has been released.

Increase the likelihood of bug and security holes.

Increase the burden of testing.

Classes designed for inheritance  should rarely implement Serializable, and interfaces should rarely extend it.

Exceptions: Throwable, Component, and HttpServlet, etc.

**Item 75: Consider using a custom serialized form**

Use the default serialized form only when an object’s phys-ical representation is identical to its logical content.

Disadvantages of inappropriate default serialized form:

* permanently ties the exported API to the  current internal representation.
* consume excessive space.
* consume excessive time (graph traversal).
* stack overflow.

Provide writeObject and readOb-ject methods implementing this serialized form.  The transient modifier indicates that an instance field is to be omitted from a class’s default serialized form.

Before deciding to make a field nontransient, convince yourself that its value is part of the logical state of the object.

Declare an explicit serial version UID in every serializable class you write, to tackle the version problem.

**Item 76: Write  readObject methods defensively**

Make defensive copy of fields in readObject() method is necessary like in the constructor.

For classes with object reference fields that must remain private, defensively copy each object in such a field.

Check any invariants and throw an  InvalidObjectException if a check fails.

Do not invoke any overridable methods in the constructor or readObject() method.

**Item 77: For instance control, prefer enum types to readResolve**

The instance control (e.g, Singleton) is violated without readResolve() method after serialization.

All instance fields with object reference types must be declared transient if using the readResolve() to do the instance control.

Instance control through Enum is preferred to the readResolve().

**Item 78: Consider serialization proxies instead of serialized instances**

Serialization proxy pattern is implemented based on an inner static class with a single constructor.

Use writeReplace() of enclosing class to write a proxy instance instead of the instance of enclosing class.

Use readResolve() method to return a instance of enclosing class at the time of deserialization, since the method in the inner class is able to call the constructor of enclosing class.

Two limitations of the serialization proxy pattern:

It is not compatible with classes that are extendable by their clients (in that time, the enclosing class has not been initialized).

It is not compatible with some classes whose object graphs contain circularities.

Serialization proxy pattern is more secure, but with higher expense.