Effective Programming

# Creating & Destroying Objects

## Consider Static Factory Methods instead of constructors

A class can provide its clients with static factory methods **instead of**, or **in addition to**, public constructors.

* **Advantages:**

**One advantage** of static factory methods is that, unlike constructors, they have names. with a well-chosen name is easier to use and the resulting client code easier to read. for a probably prime number for example:

BigInteger(int, int, Random) vs BigInteger.probablePrime

A class can have only a single constructor with a given signature. Programmers have been known to get around this restriction by providing two constructors whose parameter lists differ only in the order of their parameter types. This is a really bad idea. Instead, you can use static factory methods with carefully-chosen names.

**A second advantage** of static factory methods is that, unlike constructors, they are not required to create a new object each time they’re invoked. This allows immutable classes to use preconstructed instances, or to cache instances as they’re constructed, and dispense them repeatedly to avoid creating unnecessary duplicate objects. The Boolean.valueOf(boolean) method illustrates this technique: it *never* creates an object**. It can greatly improve performance if equivalent objects are requested often**, especially if they are expensive to create. The ability of static factory methods to return the same object from repeated invocations, allows classes to maintain strict control over what instances exist at any time. Classes that do this are said to be ***instance-controlled****.* There are several reasons to write instance-controlled classes. Instance control allows a class to guarantee that it is a singleton or non-instantiable. Also, it allows an immutable value class to make the guarantee that no two equal instances exist: a.equals(b), if and only if a==b. This is the basis of the *Flyweight* pattern. **Enum types provide this guarantee**.

# Methods Common to All Objects

# Classes and Interfaces

## Minimize the Accessibility of Classes and Members

* A well-designed component hides all its implementation details, cleanly separating its API from its implementation. **Components then communicate only through their APIs and are oblivious to each others’ inner workings**. This concept, known **as *information hiding*****or** ***encapsulation***, is a fundamental tenet of software design. You can read about the benefits of this on the book.
* If a top-level class or interface can be made package-private, it should be. By making it package-private, you make it part of the implementation rather than the exported API, and you can modify it, replace it, or eliminate it in a subsequent release without fear of harming existing clients. If you make it public, you are obligated to support it forever to maintain compatibility.
* If a package-private top-level class or interface is used **by only one class**, consider making the top-level class **a private static nested class** of the sole class that uses it.
* After carefully designing your class’s public API, your reflex should be to make all other members private
* Only if another class in the same package really needs to access a member should you remove the private modifier, making the member package-private. If you find yourself doing this often, you should reexamine the design of your system to see if another decomposition might yield classes that are better decoupled from one another.
* Although the package-level and private members are not a part of the exported API, they can leak to the exported API if the class implements Serializable.
* For members of public classes, a huge increase in accessibility occurs when the access level goes from package-private to protected. A protected member is part of the class’s exported API and must be supported forever. The need for protected members should be relatively rare.
* If a method overrides a superclass method, it cannot have a more restrictive access level in the subclass than in the superclass. This is necessary to ensure that an instance of the subclass is usable anywhere that an instance of the superclass is usable (**the *Liskov substitution principle****).* A special case of this rule is that if a class implements an interface, all of the class methods that are in the interface must be declared public in the class.
* To facilitate testing your code, you may be tempted to make a class, interface, or member more accessible than otherwise necessary. This is fine up to a point. It is acceptable to make a private member of a public class package-private in order to test it, but it is not acceptable to raise the accessibility any higher. In other words, it is not acceptable to make a class, interface, or member a part of a pack-age’s exported API to facilitate testing.
* **Instance** fields of public classes should rarely be public. If an instance field is nonfinal or is a reference to a mutable object, then by making it public, you give up the ability to limit the values that can be stored in the field. This means you give up the ability to enforce invariants involving the field. Also, you give up the ability to take any action when the field is modified, so **classes with public mutable fields are not generally thread-safe. Even if a field is final and refers to an immutable object**, by making it public you give up the flexibility to switch to a new internal data representation in which the field does not exist.
* The same advice applies to static fields, with one exception. **You can expose constants via public static final fields**, assuming the constants form an integral part of the abstraction provided by the class. By convention, such fields have names consisting of capital letters, with words separated by underscores. It is critical that these fields contain **either primitive values or references to immutable objects**. a field containing a reference to a mutable object has all the disadvantages of a nonfinal field. While the reference cannot be modified, the referenced object can be modified—with disastrous results.(Why Static ok but instance field nokay?)
* Note that a nonzero-length array is **always mutable**, so **it is wrong for a class to have a public static final array field, or an accessor that returns such a field.** If a class has such a field or accessor, clients will be able to modify thecontents of the array. This is a frequent source of security holes(Why??)

You can either make a public static final field that is an unmodifiable list form of that array or provide a public getter which returns a copy of it: array.clone. in both cases the original field will be private.

* Summary:

To summarize, you should reduce accessibility of program elements as much as possible (within reason). After carefully designing a minimal public API, you should prevent any stray classes, interfaces, or members from becoming part of the API. With the exception of public static final fields, which serve as constants, public classes should have no public fields. Ensure that objects referenced by public static final fields are immutable.

## in Public Classes Use Accessor Methods, Not Public Fields

## Minimize Mutability

## Favor Composition Over Inheritance

## Design and Document for Inheritance or Else Prohibit it

## Prefer Interfaces to Abstract Classes

## Design Interfaces for Posterity

## Use Interfaces Only to Define Types

## Prefer Class Hierarchies to Tagged Classes

## Favor Static Class Members Over Non-Static

## Limit Source Files to a Single Top-Level Class