NSObject

Inherits From:	none *(NSObject is the root class)*
Conforms To:	NSCoding NSObject
Declared In:	foundation/NSObject.h

Class Description

NSObject is the root class of all ordinary Objective C inheritance hierarchies; it's the one class that has no superclass. From NSObject, other classes inherit a basic interface to the run-time system for the Objective C language. It's through NSObject that instances of all classes obtain their ability to behave as objects.

Among other things, the NSObject class provides inheriting classes with a framework for creating, initializing, deallocating, copying, comparing, and archiving objects, for performing methods selected at run-time, for querying an object about its methods and its position in the inheritance hierarchy, and for forwarding messages to other objects. For example, to ask an object what class it belongs to, you'd send it a **class** message (**class** is declared in the NSObject protocol). To find out whether it implements a particular method, you'd send it a **respondsToSelector:** message.

The NSObject class is an abstract class; programs use instances of classes that inherit from NSObject, but never of NSObject itself.

Initializing an Object to Its Class

Every object is connected to the run-time system through its **isa** instance variable, inherited from the NSObject class. **isa** identifies the object's class; it points to a structure that's compiled from the class definition. Through **isa**, an object can find whatever information it needs at run time—such as its place in the inheritance hierarchy, the size and structure of its instance variables, and the location of the method implementations it can perform in response to messages.

Because all objects directly or indirectly inherit from the NSObject class, they all have this variable. The defining characteristic of an "object" is that its first instance variable is an **isa** pointer to a class structure.

The installation of the class structure—the initialization of **isa**—is one of the responsibilities of the **alloc**, **allocWithZone:**, and **new** methods, the same methods that create (allocate memory for) new instances of a class. In other words, class initialization is part of the process of creating an object; it's not left to the methods, such as **init**, that initialize individual objects with their particular characteristics.

Instance and Class Methods

Every object requires an interface to the run-time system, whether it's an instance object or a class object. For example, it should be possible to ask either an instance or a class about its position in the inheritance hierarchy or whether it can respond to a particular message.

So that this won't mean implementing every NSObject method twice, once as an instance method and again as a class method, the run-time system treats methods defined in the root class in a special way:

Instance methods defined in the root class can be performed both by instances and by class objects.

A class object has access to class methods—those defined in the class and those inherited from the classes above it in the inheritance hierarchy—but generally not to instance methods. However, the run-time system gives all class objects access to the instance methods defined in the root class. Any class object can perform any root instance method, provided it doesn't have a class method with the same name.

For example, a class object could be sent messages to perform NSObject's **respondsToSelector:** and **perform:withObject:** instance methods:

```
SEL method = @selector(riskAll:);

if ( [MyClass respondsToSelector:method] )
    [MyClass perform:method withObject:self];
```

When a class object receives a message, the run-time system looks first at the receiver's repertoire of class methods. If it fails to find a class method that can respond to the message, it looks at the set of instance methods defined in the root class. If the root class has an instance method that can respond (as NSObject does for **respondsToSelector:** and **perform:withObject:**), the run-time system uses that implementation and the message succeeds.

Note that the only instance methods available to a class object are those defined in the root class. If MyClass in the example above had reimplemented either **respondsToSelector:** or **perform:withObject:**, those new versions of the methods would be available only to instances. The class object for MyClass could perform only the versions defined in the NSObject class. (Of course, if MyClass had implemented **respondsToSelector:** or

perform:withObject: as class methods rather than instance methods, the class would perform those new versions.)

Interface Conventions

The NSObject class defines a number of methods that other classes are expected to override. Often, NSObject's default implementation simply returns **self**. Putting these "empty" methods in the NSObject class serves two purposes:

- It means that every object can readily respond to certain standard messages, such as **load** or **init**, even if the response is to do nothing. It's not necessary to check (using **respondsToSelector:**) before sending the message.
- It establishes conventions that, when followed by all classes, make object interactions more reliable. These conventions are explained in full under the method descriptions.

Instance Variables

Class **isa**;

isa	A pointer to the instance's class structure.

Adopted Protocols

NSCoding	– encodeWithCoder: – initWithCoder:

NSObject	– autorelease – class – conformsToProtocol: – hash – isEqual: – isKindOfClass: – isMemberOfClass: – isProxy – perform: – perform:withObject: – perform:withObject:withObject: – release – respondsToSelector: – retain – retainCount – self – zone

Method Types

Initializing the class	+ initialize
Creating and destroying instances	+ allocWithZone: + alloc – init + new – dealloc
Identifying classes	+ class + superclass
Testing class functionality	+ instancesRespondToSelector:
Testing protocol conformance	+ conformsToProtocol:
Obtaining method information	– methodForSelector: + instanceMethodForSelector: – methodSignatureForSelector:
Describing objects	+ description – description
Posing	+ poseAsClass:
Error handling	– doesNotRecognizeSelector:

Forwarding messages	– forwardInvocation:
Dynamic loading	+ load
Archiving	– awakeAfterUsingCoder: – classForArchiver – classForCoder – replacementObjectForArchiver – replacementObjectForCoder: + setVersion: + version

Class Methods

alloc

+ **alloc**

Returns a new instance of the receiving class. The **isa** instance variable of the new object is initialized to a data structure that describes the class; memory for all other instance variables is set to 0. This method invokes **allocWithZone:**, passing the default zone (as returned by **NSDefaultMallocZone()**) as its argument.

A version of the **init** method should be used to complete the initialization process. For example:

```
TheClass *newObject = [[TheClass alloc] init];
```

Other classes shouldn't override **alloc** to add code that initializes the new instance. Instead, class-specific versions of the **init** method should be implemented for that purpose. Methods can also be implemented to combine allocation and initialization.

Note that it's your responsibility to release objects (with either **release** or **autorelease**) created with the **alloc...** methods.

See also: + **allocWithZone:**, – **init**, + **new**

allocWithZone:

+ **allocWithZone:**(NSZone *)*zone*

Returns a new instance of the receiving class. Memory for the new object is allocated from *zone*.

The **isa** instance variable of the new object is initialized to a data structure that describes the class; memory for its other instance variables is set to 0. A version of the **init** method should be used to complete the initialization process. For example:

```
TheClass *newObject = [[TheClass allocWithZone:someZone] init];
```

The **allocWithZone:** method shouldn't be overridden to include any initialization code. Instead, class-specific versions of the **init** method should be implemented for that purpose.

When one object creates another, it's often a good idea to make sure they're both allocated from the same region of memory. The **zone** method (declared in the NSObject protocol) can be used for this purpose; it returns the zone where the receiver is located. For example:

```
id myCompanion = [[TheClass allocWithZone:[self zone]] init];
```

Note that it's your responsibility to release objects (with either **release** or **autorelease**) created with the **alloc...** methods.

See also: + **alloc**, – **init**

class

+ (Class)**class**

Returns **self**. Since this is a class method, it returns the class object.

When a class is the receiver of a message, it can be referred to by name. In all other cases, the class object must be obtained through this, or a similar method. For example, here SomeClass is passed as an argument to the **isKindOfClass:** method (declared in the NSObject protocol):

```
BOOL test = [self isKindOfClass:[SomeClass class]];
```

See also: – **class** (NSObject protocol)

conformsToProtocol:

+ (BOOL)**conformsToProtocol:**(Protocol *)*aProtocol*

Returns YES if the receiving class conforms to *aProtocol*, and NO if it doesn't.

A class is said to "conform to" a protocol if it adopts the protocol or inherits from another class that adopts it. Protocols are adopted by listing them within angle brackets after the interface declaration. Here, for example, MyClass adopts the imaginary AffiliationRequests and Normalization protocols:

```
@interface MyClass : NSObject <AffiliationRequests, Normalization>
```

A class also conforms to any protocols that are incorporated in the protocols it adopts or inherits. Protocols incorporate other protocols in the same way that classes adopt them. For example, here the AffiliationRequests protocol incorporates the Joining protocol:

```
@protocol AffiliationRequests <Joining>
```

When a class adopts a protocol, it must implement all the methods the protocol declares. If it adopts a protocol that incorporates another protocol, it must also implement all the methods in the incorporated protocol or inherit those methods from a class that adopts it. In the example above, MyClass must implement the methods in the AffiliationRequests and Normalization protocols and, in addition, either inherit from a class that adopts the Joining protocol or implement the Joining methods itself.

When these conventions are followed and all the methods in adopted and incorporated protocols are in fact implemented, the **conformsToProtocol:** test for a set of methods becomes roughly equivalent to the **respondsToSelector:** test for a single method.

However, **conformsToProtocol:** judges conformance solely on the basis of the formal declarations in source code, as illustrated above. It doesn't check to see whether the methods declared in the protocol are actually implemented. It's the programmer's responsibility to see that they are.

The Protocol object required as this method's argument can be specified using the **@protocol()** directive:

```
BOOL canJoin = [MyClass conformsToProtocol:@protocol(Joining)];
```

See also: – **conformsToProtocol:**

description

+ (NSString *)**description**

Returns a string object that represents the contents of the receiver. The debugger's **print-object** command invokes this method to produce a textual description of an object.

NSObject's implementation of this method simply prints the name of the class.

See also: – **description**

initialize

+ **initialize**

Initializes the class before it's used (before it receives its first message). The run-time system sends an **initialize** message to each class just before the class, or any class that inherits from it, is sent its first message from within the program. Each class object receives the **initialize** message just once. Superclasses receive it before subclasses do.

For example, if the first message your program sends is this,

```
[Application new]
```

the run-time system will generate these three **initialize** messages,

```
[NSObject initialize];
[Responder initialize];
[Application initialize];
```

since Application is a subclass of Responder and Responder is a subclass of NSObject. All the **initialize** messages precede the **new** message and are sent in the order of inheritance, as shown.

If your program later begins to use the Text class,

```
[Text instancesRespondToSelector:someSelector]
```

the run-time system will generate these additional **initialize** messages,

```
[View initialize];
[Text initialize];
```

since the Text class inherits from NSObject, Responder, and View. The **instancesRespondToSelector:** message is sent only after all these classes are initialized. Note that the **initialize** messages to NSObject and Responder aren't repeated; each class is initialized only once.

You can implement your own versions of **initialize** to provide class-specific initialization as needed.

Because **initialize** methods are inherited, it's possible for the same method to be invoked many times, once for the class that defines it and once for each inheriting class. To prevent code from being repeated each time the method is invoked, it can be bracketed as shown in the example below:

```
+ initialize
{
    if ( self == [MyClass class] ) {
        /* put initialization code here */
    }
    return self;
}
```

Since the run-time system sends a class just one **initialize** message, the test shown in the example above should prevent code from being invoked more than once. However, if for some reason an application also generates **initialize** messages, a more explicit test may be needed:

```
+ initialize
{
    static BOOL tooLate = NO;
    if ( !tooLate ) {
        /* put initialization code here */
        tooLate = YES;
    }
    return self;
}
```

See also: – **init**, – **class** (NSObject protocol)

instanceMethodForSelector:

+ (IMP)**instanceMethodForSelector:**(SEL)*aSelector*

Locates and returns the address of the implementation of the *aSelector* instance method. An error is generated if instances of the receiver can't respond to *aSelector* messages.

This method is used to ask the class object for the implementation of an instance method. To ask the class for the implementation of a class method, use the instance method **methodForSelector:** instead of this one.

instanceMethodForSelector:, and the function pointer it returns, are subject to the same constraints as those described for **methodForSelector:**.

See also: – **methodForSelector:**

instancesRespondToSelector:

+ (BOOL)**instancesRespondToSelector:**(SEL)*aSelector*

Returns YES if instances of the class are capable of responding to *aSelector* messages, and NO if they're not. To ask the class whether it, rather than its instances, can respond to a particular message, use the **respondsToSelector:** instance method instead of **instancesRespondToSelector:**.

If *aSelector* messages are forwarded to other objects, instances of the class will be able to receive those messages without error even though this method returns NO.

See also: – **respondsToSelector:**, – **forwardInvocation:**

load

+ **load**

Implemented by subclasses to integrate the class, or a category of the class, into a running program. The NSObject implementation of this method simply returns **self**.

A **load** message is sent immediately after the class or category has been dynamically loaded into memory, but only if the newly loaded class or category implements a method that can respond.

Once a dynamically loaded class is used, it will also receive an **initialize** message. However, because the **load** message is sent immediately after the class is loaded, it always precedes the **initialize** message, which is sent only when the class receives its first message from within the program.

A **load** method is specific to the class or category where it's defined; it's not inherited by subclasses or shared with the rest of the class. Thus a class that has four categories can define a total of five **load** methods, one in each category and one in the main class definition. The method that's performed is the one defined in the class or category just loaded.

new

+ **new**

Allocates a new instance of the receiving class, sends it an **init** message, and returns the initialized object returned by **init**.

As defined in the NSObject class, **new** is a combination of **alloc** and **init**. Like **alloc**, it initializes the **isa** instance variable of the new object so that it points to the class data structure. It then invokes the **init** method to complete the initialization process.

Unlike **alloc**, **new** is sometimes reimplemented in subclasses to have it invoke a class-specific initialization method. If the **init** method includes arguments, they're typically reflected in the **new** method as well. For example:

```
+ newArg:(int)tag arg:(struct info *)data
{
    return [[self alloc] initArg:tag arg:data];
}
```

However, there's little point in implementing a **new...** method if it's simply a shorthand for **alloc** and **init...**, like the one shown above. Often **new...** methods will do more than just allocation and initialization. In some classes, they manage a set of instances, returning the one with the requested properties if it already exists, allocating and initializing a new one only if necessary. For example:

```
+ newArg:(int)tag arg:(struct info *)data
{
    MyClass *theInstance;

    if ( theInstance = findTheObjectWithTheTag(tag) )
        return theInstance;
    return [[self alloc] initArg:tag arg:data];
}
```

Although it's appropriate to define new **new...** methods in this way, the **alloc** and **allocWithZone:** methods should never be augmented to include initialization code.

See also: – **init**, + **alloc**, + **allocWithZone:**

poseAsClass

+ **poseAsClass:**(Class)*aClassObject*

Causes the receiving class to "pose as" its superclass, the *aClassObject* class. The receiver takes the place of *aClassObject* in the inheritance hierarchy; all messages sent to *aClassObject* will actually be delivered to the receiver. The receiver must be defined as a subclass of *aClassObject*. It can't declare any new instance variables of its own, but it can define new methods and override methods defined in the superclass. The **poseAsClass:** message should be sent before any messages are sent to *aClassObject* and before any instances of *aClassObject* are created.

This facility allows you to add methods to an existing class by defining them in a subclass and having the subclass substitute for the existing class. The new method definitions will be inherited by all subclasses of the superclass. Care should be taken to ensure that this doesn't generate errors.

A subclass that poses as its superclass still inherits from the superclass. Therefore, none of the functionality of the superclass is lost in the substitution. Posing doesn't alter the definition of either class.

Posing is useful as a debugging tool, but category definitions are a less complicated and more efficient way of augmenting existing classes. Posing admits only two possibilities that are absent for categories:

- A method defined by a posing class can override any method defined by its superclass. Methods defined in categories can replace methods defined in the class proper, but they cannot reliably replace methods defined in other categories. If two categories define the same method, one of the definitions will prevail, but there's no guarantee which one.

- A method defined by a posing class can, through a message to **super**, incorporate the superclass method it overrides. A method defined in a category can replace a method defined elsewhere by the class, but it can't incorporate the method it replaces.

If successful, this method returns **self**. If not, it generates an error message and aborts.

setVersion

+ (void)**setVersion:**(int)*anInt*

Sets the class version number to *anInt*, and returns **self**. The version number is helpful when instances of the class are to be archived and reused later. The default version is 0.

See also: + **version**

superclass

+ **superclass**

Returns the class object for the receiver's superclass.

See also: + **class**, – **superclass**

version

+ (int)**version**

Returns the version number assigned to the class. If no version has been set, this will be 0.

See also: + **setVersion:**

Instance Methods

awakeAfterUsingCoder:

– **awakeAfterUsingCoder:**(NSCoder *)*aCoder*

This method is implemented by subclasses to reinitialize the receiver, providing one last chance for the object to propose another in its place. The NSObject implementation of this method simply returns **self**.

This method is necessary because an object (say an instance of a subclass **Font**) may decide to replace itself after all the **initWithCoder** messages have been processed, and the method deciding to replace maybe in the middle of the inheritance hierarchy (say, **Font**).

If a replacement takes place, the implementation of **awakeAfterUsingCoder:** is responsible for releasing the old self.

See also: – **initWithCoder:**(NSCoding protocol)

classForArchiver

– (Class)**classForArchiver**

Identifies the class to be used during archiving. NSObject's implementation returns the object returned by **classForCoder:**.

classForCoder

– (Class)**classForCoder**

Identifies the class to be used during coding. An NSObject returns its own class by default.

See also: – **class** (NSObject protocol)

dealloc

– (void)**dealloc**

Deallocates the memory occupied by the receiver. Subsequent messages to the object will generate an error indicating that a message was sent to a freed object (provided that the freed memory hasn't been reused yet).

You never send a **dealloc** message directly. Instead, an object's **dealloc** method is invoked indirectly through the **release** method. See the introduction to the Foundation Kit for more details on the use of these methods.

Subclasses must implement their own versions of **dealloc** to allow the deallocation of any additional memory consumed by the object—such as dynamically allocated storage for data, or other objects that are tightly coupled to the freed object and are of no use without it. After performing the class-specific deallocation, the subclass method should incorporate superclass versions of **dealloc** through a message to **super**:

```
- dealloc {
    [companion release];
    free(privateMemory);
    vm_deallocate(task_self(), sharedMemory, memorySize);
    [super dealloc];
}
```

See also: – **release** (NSObject protocol), – **autorelease** (NSObject protocol)

description

– (NSString *)**description**

Returns a string object that represents the contents of the receiver. The debugger's **print-object** command invokes this method to produce a textual description of an object.

NSObject's implementation of this method simply prints the name of the receiver's class and the hexadecimal value of its **id**.

See also: + **description**

doesNotRecognizeSelector:

– (void)**doesNotRecognizeSelector:**(SEL)*aSelector*

Handles *aSelector* messages that the receiver doesn't recognize. The run-time system invokes this method whenever an object receives an *aSelector* message that it can't respond to or forward. This method, in turn, raises an NSInvalidArgumentException exception, and generates an error message.

doesNotRecognizeSelector: messages are generally sent only by the run-time system. However, they can be used in program code to prevent a method from being inherited. For example, an NSObject subclass might renounce the **copy** method by reimplementing it to include a **doesNotRecognizeSelector:** message as follows:

```
- copy
{
    [self doesNotRecognizeSelector:_cmd];
}
```

This code prevents instances of the subclass from recognizing or forwarding **copy** messages—although the **respondsToSelector:** method will still report that the receiver has access to a **copy** method.

(The **_cmd** variable identifies the current selector; in the example above, it identifies the selector for the **copy** method.)

See also: – **forwardInvocation:**

forwardInvocation:

– (void)**forwardInvocation:**(NSInvocation *)*anInvocation*

Implemented by subclasses to forward messages to other objects. When an object is sent a message for which it has no corresponding method, the run-time system gives the receiver an opportunity to delegate the message to another receiver. It does this by creating an NSInvocation object representing the message and sending the receiver a **forwardInvocation:** message containing this NSInvocation as the argument. The receiver's **forwardInvocation:** method can then choose to forward the message to another object. (If the delegated receiver can't respond to the message either, it too will be given a chance to forward it.)

The **forwardInvocation:** message thus allows an object to establish relationships with other objects that will, for certain messages, act on its behalf. The forwarding object is, in a sense, able to "inherit" some of the characteristics of the object it forwards the message to.

A **forwardInvocation:** message is generated only if the message encoded in *anInvocation* isn't implemented by the receiving object's class or by any of the classes it inherits from.

An implementation of the **forwardInvocation:** method has two tasks:

- To locate an object that can respond to the message encoded in *anInvocation*. This need not be the same object for all messages.
- To dispatch the message to that object.

In the simple case, in which an object forwards messages to just one destination (such as the hypothetical **friend** instance variable in the example below), a **forwardInvocation:** method could be as simple as this:

```
- (void)forwardInvocation:(NSInvocation *)invocation
{
    if ([friend respondsToSelector:[invocation selector]]) {
        [invocation setTarget:friend];
        [invocation dispatch];
    }
    [self doesNotRecognizeSelector:aSelector];
}
```

The message that's forwarded must have a fixed number of arguments; variable numbers of arguments (in the style of **printf()**) are not supported.

The return value of the message that's forwarded is returned to the original sender. All types of return values can be delivered to the sender: **id**s, structures, double-precision floating point numbers.

Implementations of the **forwardInvocation:** method can do more than just forward messages. **forwardInvocation:** can, for example, be used to consolidate code that responds to a variety of different messages, thus avoiding the necessity of having to write a separate method for each selector. A **forwardInvocation:** method might also involve several other objects in the response to a given message, rather than forward it to just one.

The default version of **forwardInvocation:** implemented in the NSObject class simply invokes the **doesNotRecognizeSelector:** method; it doesn't forward messages. Thus, if you choose not to implement **forwardInvocation:**, unrecognized messages will generate an error and cause the task to abort.

See also: – **doesNotRecognizeSelector:**

hash

@protocol NSObject
– (unsigned int)**hash**

Returns an unsigned integer that can be used as a table address in a hash table structure. For NSObject, **hash** returns a value based on the object's **id**. If two objects are equal (as determined by the **isEqual:** method), they will have the same hash value.This last point is particularly important if you define **hash** in a subclass and intend to put instances of that subclass into a collection.

See also: – **isEqual:** (NSObject protocol)

init

– **init**

Implemented by subclasses to initialize a new object (the receiver) immediately after memory for it has been allocated. An **init** message is generally coupled with an **alloc** or **allocWithZone:** message in the same line of code:

```
TheClass *newObject = [[TheClass alloc] init];
```

An object isn't ready to be used until it has been initialized. The version of the **init** method defined in the NSObject class does no initialization; it simply returns **self**.

Subclass versions of this method should return the new object (**self**) after it has been successfully initialized. If it can't be initialized, they should free the object and return **nil**.

In some cases, an **init** method might free the new object and return a substitute. Programs should therefore always use the object returned by **init**, and not necessarily the one returned by **alloc** or **allocWithZone:**, in subsequent code.

Every class must guarantee that the **init** method returns a fully functional instance of the class. Typically this means overriding the method to add class-specific initialization code. Subclass versions of **init** need to incorporate the initialization code for the classes they inherit from, through a message to **super**:

```
- init
{
    if (self = [super init]) {
    /* class-specific initialization goes here */
    }
    return self;
}
```

Note that the message to **super** precedes the initialization code added in the method. This ensures that initialization proceeds in the order of inheritance.

Subclasses often add arguments to the **init** method to allow specific values to be set. The more arguments a method has, the more freedom it gives you to determine the character of initialized objects. Classes often have a set of **init...** methods, each with a different number of arguments. For example:

```
- init;
- initArg:(int)tag;
- initArg:(int)tag arg:(struct info *)data;
```

The convention is that at least one of these methods, usually the one with the most arguments, includes a message to **super** to incorporate the initialization of classes higher up the hierarchy. This method is the *designated initializer* for the class. The other **init...** methods defined in the class directly or indirectly invoke the designated initializer through messages to **self**. In this way, all **init...** methods are chained together. For example:

```
- init
{
    return [self initArg:-1];
}

- initArg:(int)tag
{
    return [self initArg:tag arg:NULL];
}

- initArg:(int)tag arg:(struct info *)data
{
    [super init. . .];
    /* class-specific initialization goes here */
}
```

In this example, the **initArg:arg:** method is the designated initializer for the class.

If a subclass does any initialization of its own, it must define its own designated initializer. This method should begin by sending a message to **super** to perform the designated initializer of its superclass. Suppose, for example, that the three methods illustrated above are defined in the B class. The C class, a subclass of B, might have this designated initializer:

```
- initArg:(int)tag arg:(struct info *)data arg:anObject
{
    [super initArg:tag arg:data];
    /* class-specific initialization goes here */
}
```

If inherited **init...** methods are to successfully initialize instances of the subclass, they must all be made to (directly or indirectly) invoke the new designated initializer. To accomplish this, the subclass is obliged to cover (override) only the designated initializer of the superclass. For example, in addition to its designated initializer, the C class would also implement this method:

```
- initArg:(int)tag arg:(struct info *)data
{
    return [self initArg:tag arg:data arg:nil];
}
```

This ensures that all three methods inherited from the B class also work for instances of the C class.

Often the designated initializer of the subclass overrides the designated initializer of the superclass. If so, the subclass need only implement the one **init...** method.

These conventions maintain a direct chain of **init...** links, and ensure that the **new** method and all inherited **init...** methods return usable, initialized objects. They also prevent the possibility of an infinite loop wherein a subclass method sends a message (to **super**) to perform a superclass method, which in turn sends a message (to **self**) to perform the subclass method.

This **init** method is the designated initializer for the NSObject class. Subclasses that do their own initialization should override it, as described above.

See also: + **new**, + **alloc**, + **allocWithZone:**

isEqual:

@protocol NSObject
– (BOOL)**isEqual:***anObject*

Returns YES if the receiver and *anObject* are equal; otherwise returns NO. For NSObject, the **id** of *anObject* and the receiver are compared to determine equality.

methodForSelector:

– (IMP)**methodForSelector:**(SEL)*aSelector*

Locates and returns the address of the receiver’s implementation of the *aSelector* method, so that it can be called as a function. If the receiver is an instance, *aSelector* should refer to an instance method; if the receiver is a class, it should refer to a class method.

aSelector must be a valid, non-NULL selector. If in doubt, use the **respondsToSelector:** method to check before passing the selector to **methodForSelector:**.

IMP is defined as a pointer to a function that returns an **id** and takes a variable number of arguments (in addition to the two “hidden” arguments—**self** and **_cmd**—that are passed to every method implementation):

```
typedef id (*IMP)(id, SEL, ...);
```

This definition serves as a prototype for the function pointer that **methodForSelector:** returns. It’s sufficient for methods that return an object and take object arguments. However, if the *aSelector* method takes different argument types or returns anything but an **id**, its function counterpart will be inadequately prototyped. Lacking a prototype, the compiler will promote **float**s to **double**s and **char**s to **int**s, which the implementation won’t expect. It will therefore behave differently (and erroneously) when called as a function than when performed as a method.

To remedy this situation, it’s necessary to provide your own prototype. In the example below, the declaration of the **test** variable serves to prototype the implementation of the **isEqual:** method. **test** is defined as pointer to a function that returns a BOOL and takes an **id** argument (in addition to the two “hidden” arguments). The value returned by **methodForSelector:** is then similarly cast to be a pointer to this same function type:

```
BOOL (*test)(id, SEL, id);
test = (BOOL (*)(id, SEL, id))[target
    methodForSelector:@selector(isEqual:)];

while ( !test(target, @selector(isEqual:), someObject) ) {
    . . .
}
```

In some cases, it might be clearer to define a type (similar to IMP) that can be used both for declaring the variable and for casting the function pointer **methodForSelector:** returns. The example below defines the **EqualIMP** type for just this purpose:

```
typedef BOOL (*EqualIMP)(id, SEL, id);
EqualIMP test;
test = (EqualIMP)[target methodForSelector:@selector(isEqual:)];

while ( !test(target, @selector(isEqual:), someObject) ) {
    . . .
}
```

Either way, it’s important to cast **methodForSelector:**’s return value to the appropriate function type. It’s not sufficient to simply call the function returned by **methodForSelector:** and cast the result of that call to the desired type. This can result in errors.

Note that turning a method into a function by obtaining the address of its implementation “unhides” the **self** and **_cmd** arguments.

See also: + **instanceMethodForSelector:**

methodSignatureForSelector:

– (NSMethodSignature *)**methodSignatureForSelector:**(SEL)*aSelector*

Returns an NSMethodSignature object that contains a description of the *aSelector* method, or nil if the *aSelector* method can’t be found. When the receiver is an instance, *aSelector* should be an instance method; when the receiver is a class, it should be a class method. This method is mostly used in the implementation of protocols.

replacementObjectForArchiver:

– (id)**replacementObjectForArchiver:**(NSArchiver *)*anArchiver*

Allows an object to substitute another object for itself during archiving. NSObject's implementation returns the object returned by **replacementObjectForCoder:**.

replacementObjectForCoder:

– **replacementObjectForCoder:**(NSCoder *)*encoder*

Allows an object to substitute another object for itself during coding. An *encoder* value of **nil** indicates that nothing should be encoded. NSObject's implementation returns **self**.