

Object-Oriented Programming

The task of composition of operations is often considered the heart of the art of programming. ... However, it will become evident that the appropriate composition of data is equally fundamental and appropriate.

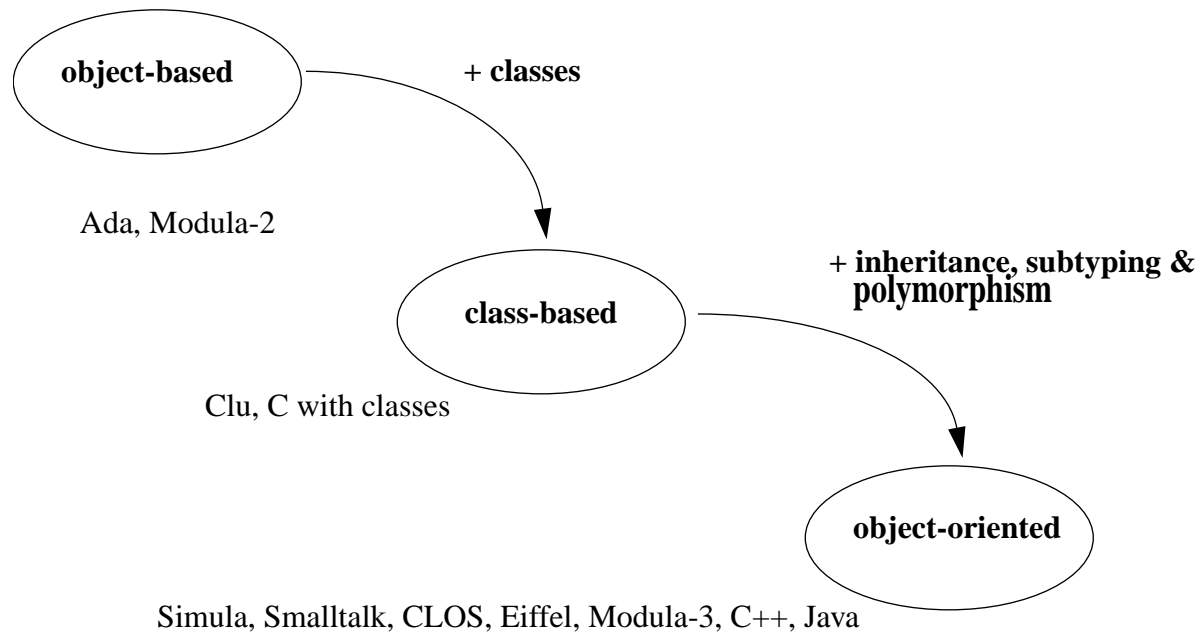
— N. Wirth, *Algorithms + Data Structures = Programs*, 1976.

Object-oriented design is based on the following concepts noted by C. A. R. Hoare, *Notes on Data Structuring*, 1972.

- **Abstraction:** the decision to concentrate on properties which are shared by many objects or situations in the real world, and to ignore the differences between them.
- **Representation:** the choice of a set of symbols to stand for the abstraction (i.e., data structures)
- **Manipulation:** the rules for transformation of the symbolic representation as a means of predicting the effect of similar manipulation in the real world (i.e., functions and procedures).
- **Axiomatization:** the rigorous statement of those properties which have been abstracted from the real world and which are shared by manipulation of the real world and the symbols which represent it.

It has taken 30+ years to bring these ideas into wide-spread use by programmers. Now, everyone wants to do object-oriented programming. Why? Object-oriented programming is now common practice among professional software engineers, and programming languages like C++ and Java allow the programmer to express solutions to computational problems in a variety of different ways. However, the important thing is not the choice of language, but being able to think about a problem in an object-oriented manner, and then use a specific language and its object-oriented features to implement a well-engineered solution that solves the problem such that a program executes **correctly, safely, and efficiently**.

Objects + Classes + Inheritance = OO Programs



Objects --- provide a conceptual framework for thinking about typed data and typed operations that correspond to a real world object that one wishes to model computationally. *The term 'object' will also be used to refer to the run-time state of a conceptual object.*

Classes --- are used to declare a new **type** with compile-time (static) attributes of data and operations (methods) that a set of objects, or *instances* of that class, will have at run-time.

Inheritance --- is a mechanism that is used to define new (sub-)types by specializing existing types. Inheritance is a compile-time (static) mechanism in a language like Java.

Polymorphism -- allow parameterization of template classes/methods by other types.

Some General OOP Terminology

An **object** is a run-time entity that represents an *encapsulation* of data, and operations defined on that data. An object has a unique identity (i.e., a memory address) and a persistent state (i.e., a set of values associated with the data locations that make up the object).

At run-time, we invoke operations on objects, called **methods**, to query the state of the object or to effect a change of state, which persists after completion of the operation. Hence, most methods defined on an object are *side-effecting* procedures. An object is created at run-time by invoking a special method called a **constructor**. In some OO languages (e.g., Java), methods may throw **exceptions** on the occurrence of an error, which must be caught and handled by the caller.

A **class** is a compile-time entity that provides a specification of an abstract data type. The specification consists of variable declarations and method declarations defined with the *scope* of the 'class' syntactic construct. In general, variable declarations are considered to be **private** and method declarations are considered to be **public**. However, variables can be declared public and methods can be declared private. Hence, a class is a linguistic mechanism used for defining new types from existing types: either builtin types, such as 'int', or previously defined class types, such as String. By also specifying the access policy (i.e., public, private) for variables and methods, a class allows the specification of information hiding properties for objects of that class.

Every object is an **instance** of some class, and hence every object is governed (constrained) by its type. Most OO languages are statically typed, so that type errors are detected at compile-time. The variables declared in a class that are associated with an object at run-time are called **instance variables**, and each object has its own unique storage locations associated with its set of instance variables. A class may define **class variables**, which are variables associated with run-time storage locations that are shared by all instances of that class.

What does it mean to program with Java?

Java is an object-oriented programming language. This means that computation is achieved via **interacting objects**. Objects interact by invoking operations on one another. Operations effect local state changes on objects, hopefully leading to some useful computation, the results of which are typically presented as output to some device.

The operations associated with a particular object must be well defined via the object's **interface**. The interface effectively defines a *contract* between the user of the object (i.e., some other object in the program) and the implementer of the object (i.e., the programmer). So, there are two perspectives to every object in an object-oriented programmer: The “user view” and the “implementer view.”

The interface, or contract, for a particular category of objects is defined using the **class** syntactic construct. A Java source program consists entirely of a set of classes that completely specify a set of object interfaces and their implementation at compile-time. A Java run-time program consists of a set of compiled bytecodes representing dynamic instantiations of those classes in the form of run-time objects that execute some computation when the bytecodes are interpreted by a Java virtual machine. The task of a Java programmer is to define new classes in terms of existing classes and builtin types. This consists of writing **class definitions**, which define new types of objects, the specification of variables related to the implementation of the public methods that provide the interface for an object, and possibly other variables and methods that are internal (private) to the implementation.

So, in Java, a program consists of lots of different class definitions. Some have to be written from scratch, others are simply reused by importing their definitions from pre-defined **packages** of class definitions (i.e., libraries). Designing a computation as a collection of compile-time class definitions each representing a category of objects that interact at run-time to perform computation is called object-oriented programming.

Naming Things: Packages, Classes, Fields and Methods

Java requires that for every class, instance variable, and method you define, that you declare an access qualifier as part of the definition, or accept the default access (which is not private access) Compare this to what you must write in C++. A public Java class must correspond to a file of the same name. So, the Hello class must be defined in a Hello.java file, which is part of the 'test' package. Technically, the main method is called:

```
edu.utexas.cs.example.Hello.main() // a fully qualified method name
```

NOTE: Unlike C++ which uses the '::' operator for scope resolution, Java uses the '.' operator for both scope resolution and class member access. A *fully qualified name* is of the form <package>.<class>.<field or method>. You might organize your packages into subpackages, in which case you write <package>.<subpackage>.<class>.<field or method>. A fully qualified name corresponds to a directory path in the filesystem, which is how Java is able to guarantee unique naming. So the name test.Hello might correspond to a file:

```
/home/user/edu/utexas/cs/example/Hello.java
```

A single .java file may contain multiple class definitions for a package, but only one may be public. The name of the public class must correspond to the name of the file. The java compiler will compile a .java file with multiple classes into multiple .class files: one for each class. The CLASSPATH environment variable is used to tell the java interpreter where to look for packages and classes. Rather than always using a fully qualified name all the time, you can "open" a package using the **import** statement. When you specify an import statement, it opens the package, and allows you to omit the package name qualifiers. A package name may be a compound name: For example, the following input/output (io) subpackage is part of the standard Java SDK package:

```
import java.io.*;
```

The import statement causes the java.applet package to be opened, and the '*' says to import all classes from that package.

How do we program in Java?

We take the “user view” and study existing packages of class definitions to learn how to interact with pre-defined categories of objects. We also take the “implementor view” and write new packages of class definitions that define new types of objects and specifies how they interact with other objects.

A **Java program** is then constructed by instantiating objects and causing them to invoke operations on one another to effect a useful computation.

Since classes are the key to understanding pre-defined object categories and for writing new ones, we have to master the structure, syntax, and object-oriented programming conventions used by Java classes.

data	instance variables and class variables
operations	object constructors object methods class methods
optional things	optional class variable initialization block
	optional nested class definition(s)

Java Data Types

Java has three categories of data types: **primitive types**, **arrays**, and **reference types**.

Because java programs execute on a virtual machine, the primitive types are defined to be the same across all real machine architectures. Note that this is different from C/C++, where the real machine architecture has an impact on the precision. Q: What is the max n for fact(n) in C/C++?

Variables defined as a primitive **boolean** type default to **false** on declaration within a class (i.e., heap allocated), but must be explicitly initialized within methods (i.e., stack allocated):

boolean	true or false
---------	---------------

Variables defined as one of the following primitive integral types default to a value of zero on declaration with a class (i.e., heap allocated), but must be explicitly initialized within methods (i.e., stack allocated):

byte	-128..127
short	-32768..32767
int	-2147483648..2147483647
long	-922337203685477508..9223372036854775807
char	0..65535

Java Data Types

Characters in java are **Unicode** characters (16 bits).

The Java String class implements a Unicode character string object, so strings can contain non-ASCII character for international character sets (e.g., ISO-8859-1, Kanji, etc.). Sorting of strings uses the Unicode collating sequences

This is a very important feature in software for today, since software must be written to permit internalization of character values and strings. For example, so that multi-lingual error message catalogs can be used.

The 7-bit ASCII character set occupies the first 128 characters of the Unicode character set, so conversions to/from ASCII are straightforward.

One important “gotcha” that arises from using Unicode strings is that character set conversions are common operations. For example, most Internet protocols (e.g., SMTP, NNTP, HTTP) use ASCII, so it is necessary to convert from/to Java String objects and byte arrays when sending/receiving data over a network socket.

Java’s floating point types implement IEEE-754 single-precision (32-bit) and double precision (64-bit) values. They default to 0.0f or 0.0d when declared in a class, but must be explicitly initialized when declared in a method:

```
float  
double
```

Objects

All objects by default *inherit* from a base object class called **java.lang.Object**. So, in Java all objects are **subtypes** of the type **Object**. That is, anywhere a type Object is called for (e.g., in a type signature of some procedure), ANY object can be used since all objects are of type Object.

This means that objects of any type can all be treated generically as instance of class Object, which is useful for defining *heterogeneous collections* of objects, such as a vector, stack, etc. In fact, the `java.util.vector` and `java.util.stack` classes are both defined to operate on objects of type Object.

Java objects are allocated from the heap and a garbage collector takes care of reaping objects from the heap when there are no more references to the object. When you create a new object, the result of the **new** operator is a reference to the heap allocated object that is assigned to an object reference variable. You use this object reference variable to access the public fields and methods of an object.

The Object class provides some high-level methods that can be called on an object of any type. For example, the Object class defines a clone method that makes a copy of an object:

You can copy objects using a special method called `java.lang.Object.clone()`, which is defined for all objects since all objects inherit from the `java.lang.Object`.

```
Vector v = new Vector();  
Vector z = v;  
Vector c = z.clone();
```

The `java.lang.Object.clone()` method does a field by field copy of the object, or what is called a “shallow” copy. It does not do a “deep” copy.

The Object Class

```
package java.lang;
/**
 * Class Object is the root of the class hierarchy.
 * Every class has Object as a superclass. All objects,
 * including arrays, implement the methods of this class.
 * @author unascribed
 * @version 1.39, 01/20/97
 * @see java.lang.Class
 * @since JDK1.0
 */

public class Object {
    public final native Class getClass();
    public native int hashCode();
    public String toString();
    public boolean equals(Object obj) { return (this == obj); }
    protected native Object clone() throws CloneNotSupportedException;
    public final native void notify();
    public final native void notifyAll();
    public final native void wait(long timeout) throws InterruptedException;
    public final void wait(long timeout, int nanos) throws InterruptedExcep-
tion { ... }
    public final void wait() throws InterruptedException { wait(0); }
    protected void finalize() throws Throwable { }
}
```

A typical Java class schemata

```
package package-name;
public class class-name {
    // instance variable declarations, with optional inline initialization
    private type-name variable-name;
    private type-name variable-name = initialization-expr;

    // class variable declaration of a constant with inline initialization
    private final static type-name variable-name = initialization-expr;

    // object constructors
    public class-name () { stmt_1; ...; stmt_n }
    public class-name ( formal-parameters ) { stmt_1; ...; stmt_n; }

    // object method definition
    public return-type method-name ( formal-parameters ) {
        stmt_1; ...; stmt_n;
    }
    // class method definition
    public static return-type method-name ( formal-parameters ) {
        stmt_1; ...; stmt_n;
    }
}
```

Type-names and return-types are taken from the set of builtin types, and previously defined class-names. A special return-type is **void** (the “un-type”). Formal parameters are a (possibly empty) comma-separated-list of name pairs: *type-name* *variable-name*.

A fragment of a typical class definition

```
public class String {
    // private instance variables, each object has its own copy
    private char value[];
    private int offset = 0;
    private int count;

    // private class variable, shared by all String objects
    private static final long serialVersionUID = -6849794470754667710L;

    // constructors
    public String() { value = new char[0]; }

    public String(char value[]) {
        this.count = value.length;
        this.value = new char[count];
        System.arraycopy(value, 0, this.value, 0, count);
    }

    // public object method
    public int length() { return count; }

    // public class method
    public static String valueOf(int i) { return Integer.toString(i, 10); }
    ...
}
```

Use of 'this' in methods

```
public class StringBuffer {
    private char value[];
    private int count;
    private boolean shared;

    public StringBuffer() { this(16); }
    public StringBuffer(int length) {
        this.value = new char[length];
        this.shared = false;
    }
    public StringBuffer(String str) {
        this(str.length() + 16);
        append(str);
    }
    public StringBuffer append(String str) {
        if (str == null) str = String.valueOf(str);
        int len = str.length();
        ensureCapacity(this.count + len);
        copyWhenShared();
        str.getChars(0, len, this.value, this.count);
        this.count += len;
        return this;
    }
}
```

NOTE: this is not defined in a class method (i.e., a method declared static)

Reference variables and object instantions

Objects are used in Java by declaring and using reference variables that are initialized using constructors. Constructors are special class methods that do not return any value.

```
class-name variable-name ;  
class-name variable-name = null ;  
class-name variable-name = new class-name ( optional-argument-list ) ;  
class-name variable-name = initialization-expression ;
```

Examples:

```
String a ;           // uninitialized String reference variable  
String b = null ;   // explicitly setting a null object reference  
String c = new String() ; // the "empty" string object  
String d = new String ("This is a string literal") ;  
String e = s ;     // initialize e using String object s ;  
a = e ;           // initialize a to refer to the same String object as e
```

Strings have a special kind of initialization expression, involving string literals.

```
String f = "this is a string literal" ;  
String g = "Hello" + ", " + "world" ;
```

These are equivalent to the initialization expressions:

```
String f = new String ("this is a string literal") ;  
String g = new String ("Hello, world") ;
```

Object and class method invocations

Once an object of a class has been instantiated and a reference variable to that object is initialized, the object is typically used in some expression as the target for some kind of method invocation using the reference variable.

```
reference-var . method-name ( optional-argument-list );
```

For example:

```
String hello = new String("Hello, world");  
int x = hello.length();
```

Class methods, which are globally accessible operations defined on classes not on objects, are invoked similarly, but using a class name instead of a reference variable.

```
class-name . class-method-name ( optional-arguments );
```

For example, an initialization expression can invoke a method that constructs an object of the appropriate type to perform the initialization.

```
String five = String.valueOf(5); // String class method to "stringify" an int
```

The `String.valueOf(int)` method is a class method that constructs a new `String` object by converting an integer value to a `String` value, assuming a base 10 radix.

Some special class definition forms

```
public class class-name {  
  
    // publicly accessible class variables  
    public static type-name variable-name = initialization-expr;  
    public static type-name variable-name;  
  
    // a class variable initialization block  
    static {  
        stmt_1; ...; stmt_n;  
    }  
  
    // special "main" class method used as a program entry point  
    public static void main (String[] args) {  
        stmt_1; ...; stmt_n;  
    }  
}
```

Class variables are very often declared public and are referenced using an expression of the form:

```
class-name . class-variable-name
```

For example, the class `java.lang.Math` class defines a public class variable representing a 64-bit approximation of π defined as `PI = 3.14159265358979323846` and used as follows:

```
double area = Math.PI * radius * radius;
```

Static Data Initialization

Static data is usually initialized at the point of declaration within a class:

```
class Program {
    static String errorMessage = "fatal error";
    static int errorCode = -1;
    ...
}
```

Alternatively, you can use a **static block** when the initialization is more complicated.

```
class Program {
    static String[] errMessages;
    static int errorCode;

    static {
        errorCode = -1;
        errMessages = new String[10];
        errMessages[0] = "fatal error";
        ...
    }
    ...
}
```

A static block is NOT the same thing as a constructor. Static variables are initialized when either a static method first is called on the class that contains the static data, or when the first object of the class is first instantiated.

Using class variables and methods

Class variables and methods can be used before any objects are ever instantiated in a Java program. The keyword **static** is used to denote class variables and methods because they are considered to be “statically” instantiated attributes of the class, in contrast to objects, which are all dynamically instantiated.

```
class Hello {
    public static String username = null;

    static {
        try {
            username = System.getProperty("user.name");
        } catch (Exception e) { System.err.println(e); }
    }

    public static void main (String[] args) {
        System.out.println("Username is: " + username);
    }
}
```

A static initialization block is typically used to initialize class variables that require more than a simple inline initialization expression, or simply to reduce “declaration clutter” and collect together all the static initializations in one place. The special class method **main** is used as the starting point for a program. It is possible for multiple classes in the same Java program to have distinct main methods. The main method called to start a Java program is determined by which class the Java VM is asked to load first.

More special class definition forms

```
public final class class-name {  
  
    // public, globally accessible constant declaration  
    public final static type-name variable-name = initialization-expr;  
  
    // a "final" object method, which cannot be overridden by a subclass  
    public final return-type method-name ( formal-parameters ) {  
        stmt_1; ...; stmt_n;  
    }  
  
    // a "native" object method, which has an external implementation  
    public native return-type method-name ( formal-parameters );  
    public final native return-type method-name ( formal-parameters );  
}
```

A final class cannot be extended. That is to say, a new class cannot be defined as an extension of a final class.

A final static variable is the same thing as a constant. A final method is one that cannot be overridden by a subclass. In a final class, all methods are implicitly final. In a non-final class, methods can be selectively made final. Only object methods can be final. Static methods are implicitly final.

A native method consists of just a method declaration with no method body. The method implementation is typically written in C/C++ and is linked into the Java run-time, usually for efficiency reasons.

Example usage of “final” and “native” qualifiers

```
public final class Math {  
  
    public static final double E = 2.7182818284590452354;  
    public static final double PI = 3.14159265358979323846;  
  
    public static native double sin (double a);  
    public static native double cos (double a);  
    public static native double tan (double a);  
    ...  
    public static native double exp(double a);  
    public static native double log(double a);  
    public static native double sqrt(double a);  
    public static native double pow(double a, double b);  
    ...  
    public static int abs(int a) { return (a < 0) ? -a : a; }  
    public static long abs(long a) { return (a < 0) ? -a : a; }  
    public static float abs(float a) { return (a < 0) ? -a : a; }  
    public static double abs(double a) { return (a < 0) ? -a : a; }  
    ...  
}
```

Note that it is necessary to **overload** the absolute value method for each the builtin numeric types. In Java, you can define many methods with the same name, as long as the type signatures are unique.

Declaration of methods that throw exceptions

```
public class class-name {  
    ...  
    // a "final" object method, which cannot be overridden by a subclass  
    public return-type method-name ( formal-parameters )  
    throws exception-type-list  
    {  
        stmt_1; ...; stmt_n;  
    }  
};
```

An exception-type-list is a comma-separated list of previously defined exception types (i.e., class names). In general, you should invoke any method that will throw an exception inside the context of a **try block**, but not necessarily in the same scope level. A catch block is defined for each specific exception type that you want to catch. An optional finally block can be defined to contain a set of statements that will be executed whether or not the exception occurs.

```
try {  
    ...; variable-name.method-name (arguments); ...;  
}  
catch (exception-type1 e1) { stmt_1; ...; stmt_n; }  
catch (exception-type2 e2) { stmt_1; ...; stmt_n; }  
...  
finally {  
    stmt_1; ...; stmt_n;  
}
```

Example: a method that throws an exception

```
public final class Byte {
    public static final byte    MIN_VALUE = -128;
    public static final byte    MAX_VALUE = 127;

    public static byte parseByte(String s) throws NumberFormatException {
        return parseByte(s, 10);
    }
    public static byte parseByte(String s, int radix)
throws NumberFormatException {
        int i = Integer.parseInt(s, radix);
        if (i < MIN_VALUE || i > MAX_VALUE)
            throw new NumberFormatException();
        return (byte)i;
    }
}

public static void main (String[] args) {
    int err_code = 0;
    try {
        byte b = Byte.parseByte(args[0]);
    }
    catch (ArrayIndexOutOfBoundsException e) { err_code = -1; }
    catch (NumberFormatException e) { err_code = -2; }
    finally { System.exit(err_code); }
}
```

Composition vs Inheritance

A fundamental design choice that we have to make when considering whether or not to use inheritance is to examine the relationship that exists between two classes. The simple way to do this is the ‘Is-a’ and ‘has-a’ rule. We say an object X is-a Y, if everywhere you can use an object of type Y, you can use instead of object of type X. In other words, X is a **proper subtype** of Y. Usually, this means that X implements the same interface as Y. So, you know that X and Y conform to the same set of method type signatures, however, their implementation may be different. We say an object X has-a Y, if Y is a part-of X. So, you typically think of X containing an instance of Y, not X inheriting from Y. For example:

1. Would you say that a Stack is-a Vector or a Stack has-Vector?
2. Would you say that Circle is-a Shape or a Circle has-a Shape?

In each case, you need to consider whether or not the relationship between two objects is simple one of “using the object” or “being the object”. If you just need to use an object, then that implies a composition relationship. If an object behaves like another object, then that implies a subtype relationship.

Subtyping, and subtype polymorphism, are one of the most important contributions of object-oriented programming to programming language theory in general. Java implements a very general form of subtype polymorphism, because every object is-a Object, since every class inherits implicitly from the class Object. You can always treat any object in Java, not matter what its class type is, as a subtype of Object. However, this is a very generic way to treat ALL objects in a program, which may in fact be too general, as it forces the programmer to do a lot of **upcasting** and **downcasting**. You cast up from a concrete type to a more general type and you cast down from a more general type of a more concrete type. Improper downcasting can however result in run-time type errors.

Inheritance in Java

Inheritance is a compile-time mechanism in Java that allows you to extend a class (called the **base class** or **superclass**) with another class (called the **derived class** or **subclass**). In Java, inheritance is used for two purposes:

1. **class inheritance** - create a new class as an extension of another class, primarily for the purpose of **code reuse**. That is, the **derived class** inherits the public methods and public data of the **base class**. Java only allows a class to have one immediate base class, i.e., single class inheritance.
2. **interface inheritance** - create a new class to implement the methods defined as part of an **interface** for the purpose of **subtyping**. That is a class that implements an interface “conforms to” (or is constrained by the type of) the interface. Java supports multiple interface inheritance.

In Java, these two kinds of inheritance are made distinct by using different language syntax. For class inheritance, Java uses the keyword **extends** and for interface inheritance Java uses the keyword **implements**.

```
public class derived-class-name extends base-class-name {  
    // derived class methods extend and possibly override  
    // those of the base class  
}
```

```
public class class-name implements interface-name {  
    // class provides an implementation for the methods  
    // as specified by the interface  
}
```

Example of class inheritance

```
package MyPackage;

class Base {
    private int x;
    public int f() { ... }
    protected int g() { ... }
}

class Derived extends Base {
    private int y;
    public int f() { /* new implementation for Base.f() */ }
    public void h() { y = g(); ... }
}
```

In Java, the **protected** access qualifier means that the protected item (field or method) is visible to a any derived class of the base class containing the protected item. It also means that the protected item is visible to methods of other classes in the same package. This is different from C++.

Q: What is the base class of class Object? I.e., what would you expect to get if you executed the following code?

```
Object x = new Object();
System.out.println(x.getClass().getSuperclass());
```

Order of Construction under Inheritance

Note that when you construct an object, the default base class constructor is called implicitly, before the body of the derived class constructor is executed. So, objects are constructed top-down under inheritance. Since every object inherits from the `Object` class, the `Object()` constructor is always called implicitly. However, you can call a superclass constructor explicitly using the builtin **super** keyword, as long as it is the *first* statement in a constructor.

For example, most Java exception objects inherit from the `java.lang.Exception` class. If you wrote your own exception class, say `SomeException`, you might write it as follows:

```
public class SomeException extends Exception {

    public SomeException() {
        super(); // calls Exception(), which ultimately calls Object()
    }

    public SomeException(String s) {
        super(s); // calls Exception(String), to pass argument to base class
    }

    public SomeException (int error_code) {
        this("error"); // class constructor above, which calls super(s)
        System.err.println(error_code);
    }
}
```

Abstract Base Classes

An **abstract class** is a class that leaves one or more method implementations unspecified by declaring one or more methods *abstract*. An abstract method has no body (i.e., no implementation). A subclass is required to override the abstract method and provide an implementation. Hence, an abstract class is incomplete and cannot be instantiated, but can be used as a base class.

```
abstract public class abstract-base-class-name {  
    // abstract class has at least one abstract method  
  
    public abstract return-type abstract-method-name ( formal-params );  
  
    ... // other abstract methods, object methods, class methods  
}  
  
public class derived-class-name extends abstract-base-class-name {  
  
    public return-type abstract-method-name (formal-params) { stmt-list; }  
  
    ... // other method implementations  
}
```

It would be an error to try to instantiate an object of an abstract type:

```
abstract-class-name obj = new abstract-class-name(); // ERROR!
```

That is, operator `new` is invalid when applied to an abstract class.

Example abstract class usage

```
abstract class Point {  
    private int x, y;  
    public Point(int x, int y) { this.x = x; this.y = y; }  
    public void move(int dx, int dy)  
        { x += dx; y += dy; plot(); }  
    public abstract void plot(); // has no implementation  
}
```

```
abstract class ColoredPoint extends Point {  
    private int color;  
    protected public ColoredPoint(int x, int y, int color)  
        { super(x, y); this.color = color; }  
}
```

```
class SimpleColoredPoint extends ColoredPoint {  
    public SimpleColoredPoint(int x, int y, int color) { super(x,y,color); }  
    public void plot() { ... } // code to plot a SimpleColoredPoint  
}
```

Since ColoredPoint does not provide an implementation of the plot method, it must be declared abstract. The SimpleColoredPoint class does implement the inherited plot method. It would be an error to try to instantiate a Point object or a ColoredPoint object. However, you can declare a Point reference and initialize it with an instance of a subclass object that implements the plot method:

```
Point p = new SimpleColoredPoint(a, b, red); p.plot();
```

Interfaces

An abstract class mixes the idea of mutable data in the form of instance variables, non-abstract methods, and abstract methods. An abstract class with only static final instance variables and all abstract methods is called an **interface**. An interface is a *specification*, or contract, for a set of methods that a class that implements the interface must conform to in terms of the type signature of the methods. The class that implements the interface provides an implementation for each method, just as with an abstract method in an abstract class.

So, you can think of an interface as an abstract class with all abstract methods. The interface itself can have either public, package, private or protected access defined. All methods declared in an interface are implicitly abstract and implicitly public. It is not necessary, and in fact considered redundant to declare a method in an interface to be abstract.

You can define data in an interface, but it is less common to do so. If there are data fields defined in an interface, then they are implicitly defined to be:

- public.
- static, and
- final

In other words, any data defined in an interface are treated as public constants.

Note that a class and an interface in the same package cannot share the same name.

Methods declared in an interace cannot be declared final. **Why?**

Interface declaration

Interface names and class names in the same package must be distinct.

```
public interface interface-name {  
  
    // if any data are defined, they must be constants  
    public static final type-name var-name = constant-expr;  
  
    // one or more implicitly abstract and public methods  
    return-type method-name ( formal-params );  
}
```

An interface declaration is nothing more than a specification to which some class that purports to implement the interface must conform to in its implementation. That is, a class that implements the interface must have defined implementations for each of the interface methods.

The major reason for interfaces being distinct elements in Java is that you often want to define some operation to operate on objects that all conform to the same interface. So, you can define some code in a very general way, with a guarantee that by only using the methods defined in the interface, that all objects that implement the interface will have defined implementations for all the methods.

For example, you might define a Shape interface that defines an area() method, and so you would expect that any class that implements the Shape interface, would define an area method. So, if I have a list of references to objects that implement the Shape interface, I can legitimately invoke the area method, abstractly, on each of these objects and expect to obtain as a result a value that represents the area of some shape.

Separation of Interface from Implementations

Interfaces are specifications for many possible implementations. Interfaces are used to define a contract for how you interact with an object, independent of the underlying implementation. The objective of an object-oriented programmer is to separate the specification of the interface from the hidden details of the implementation.

Consider the specification of a common LIFO stack.

```
public interface StackInterface {  
    boolean empty();  
    void push(Object x);  
    Object pop() throws EmptyStackException;  
    Object peek() throws EmptyStackException;  
}
```

Note that the methods in this interface are defined to operate on objects of type `Object`. Since a stack is a “container type”, it is very common to use the base class for all objects, namely `Object`, as the type of the arguments and results to a container type. Since all objects ultimately inherit from class `Object`, we can always generically refer to any object in the system using an `Object` reference. However, when we pop from a stack, we have to explicitly type case from the very general type `Object` to a concrete type, so that we can manipulate the object concretely.

Q: How many different ways are there to implement a stack? If we are using just using a `Stack` object (as opposed to implementing it ourselves) should we care?

Stack implementation of the StackInterface

```
public class Stack implements StackInterface {  
  
    private Vector v = new Vector(); // use java.util.Vector class  
  
    public boolean empty() { return v.size() == 0; }  
  
    public void push(Object item) { v.addElement(item); }  
  
    public Object pop() {  
        Object obj = peek();  
        v.removeElementAt(v.size() - 1);  
        return obj;  
    }  
  
    public Object peek() throws EmptyStackException {  
        if (v.size() == 0)  
            throw new EmptyStackException();  
        return v.elementAt(v.size() - 1);  
    }  
}
```

Should a stack use or inherit from a vector?

The `java.util.Stack` class is defined as a subclass of the `java.util.Vector` class, rather than using a `Vector` object as in the previous example. This sort of inheritance is not subtype inheritance, because the interface of a `Stack` object can be violated because a `Vector` has a “wider” interface than a `Stack`, i.e., a vector allows insertion into the front and the rear, so it is possible to violate the stack contract by treating a stack object as a vector, and violating the LIFO specification of a stack.

```
public class Stack extends Vector {
    public Object push(Object item) {addElement(item); return item;}
    public Object pop() {
        Object obj;
        int len = size();
        obj = peek();
        removeElementAt(len - 1);
        return obj;
    }
    public Object peek() {
        int len = size();
        if (len == 0) throw new EmptyStackException();
        return elementAt(len - 1);
    }
    public boolean empty() { return size() == 0;}
}
```

```
Vector v = new Stack(); // legal - base class reference to subclass object
v.insertElementAt(x, 2); // insert object x into Vector slot 2!!
```

When to use an Interface vs when to use an abstract class

Having reviewed their basic properties, there are two primary differences between interfaces and abstract classes:

- an abstract class can have a mix of abstract and non-abstract methods, so some default implementations can be defined in the abstract base class. An abstract class can also have static methods, static data, private and protected methods, etc. In other words, a class is a class, so it can contain features inherent to a class. The downside to an abstract base class, is that since there is only single inheritance in Java, you can only inherit from one class.
- an interface has a very restricted use, namely, to declare a set of public abstract method signatures that a subclass is required to implement. An interface defines a set of type constraints, in the form of type signatures, that impose a requirement on a subclass to implement the methods of the interface. Since you can inherit multiple interfaces, they are often a very useful mechanism to allow a class to have different behaviors in different situations of usage by implementing multiple interfaces.

It is usually a good idea to implement an interface when you need to define methods that are to be explicitly overridden by some subclass. If you then want some of the methods implemented with default implementations that will be inherited by a subclass, then create an implementation class for the interface, and have other class inherit (extend) that class, or just use an abstract base class instead of an interface.

Example of default interface implementations

```
interface X {
    void f();
    int g();
}

class XImpl implements X {
    void g() { return -1; } // default implementation for g()
}

class Y extends XImpl implements X {
    void f() { ... } // provide implementation for f()
}
```

Note that when you invoke an abstract method using a reference of the type of an abstract class or an interface, the method call is dynamically dispatched.

```
X x = new Y();
x.f();
```

The call `x.f()` causes a run-time determination of the actual type of object that 'x' refers to, then a method lookup to determine which implementation of `f()` to invoke. In this case, `Y.f(x)` is called, but the type of `x` is first converted to `Y` so that the 'this' reference is initialized to a reference of type `Y` inside of `f()`, since the implicit type signature of `Y.f()` is really `Y.f(final Y this)`;

Base Class Finalization

The Object class defines a special “finalize” method that is called by the garbage collector to allow an object to finalize any cleanup that needs to occur before the memory resources for the object are reclaimed. So, the finalize method, which can be overridden by an class, provides a type of destructor. The finalize method should be implemented for any class that uses system resources and needs to release those resources as part of implicit destruction by the garbage collector. For example:

```
public class DerivedFile extends BaseFile {
    private RandomAccessFile file;
    public DerivedFile(String path) {... /* do initialization */ }
    public void close() {
        if (file != null) {
            try { file.close(); file = null; } // help gc to collect object
            catch (IOException e) { ... }
        }
    }
    public void finalize() throws Throwable {
        close(); // ensure that file was closed before we die
        super.finalize(); // explicitly call BaseFile.finalize()
    }
}
```

The finalize method should always call the superclass finalize method as the last thing it does, so that the superclass has a chance to finalize itself. This is very much like the virtual destructor chain in C++, except that in Java, you have to explicitly initiate the call to the superclass finalize method. The garbage collector will call the first finalize method for you, and then you have to program the rest of the finalization calls.

Polymorphism

As we have already seen, the `Object` class is the base class for every object. This idea of a common base class `Object` is borrowed from Smalltalk. The `Object` class permits a form of **subtype polymorphism** in Java that is not found in C++. Since every object “is-a” `Object`, then it is possible to define heterogeneous collections of objects. For example, the `java.util.Vector` class, is implemented as a resizable array of type `Object`. That means that you can insert any type of java object into a single `Vector`. However, when you do this, you “lose” the type of the object. For example:

```
Vector v = new Vector();
v.addElement(new String("hello, world"));
v.addElement(new Integer(5));
Object s = v.firstElement();
Object i = v.lastElement();
```

In this simple example, you have to know at compile-time that ‘s’ is really a reference to an object of type `String`, and ‘i’ is a reference to an element of type `Integer`. What you really want to write is an explicit downcast:

```
String s = (String) v.firstElement();
Integer i = (Integer) v.lastElement();
```

Polymorphism in Java permits this type of explicit **downcasting** of an `Object` reference to a reference of the object’s real type. If you get it wrong, the java run-time will raise an exception called the `ClassCastException`. This is a case where you should use the **instanceof** language feature to check the real type of a subtype of the `Object` class at runtime before performing the type cast if you do not know for sure that you are casting to the correct type.

Finding the Class of an Object at Runtime

Because every instance of a class is also an instance of the Object class, you can use an Object reference to determine at run-time the class of an object. In fact, Java defines a class called “java.lang.Class” that allows you to do some interesting things that are not easily achievable in C++. For example, you can query an object to determine its type:

```
Object x = vector.removeLast();
Class type = x.getClass(); // get the class of the thing that was in the vector
System.out.println(type.getName());
```

You can then ask the Class object for certain kinds of information about the class for an object, such as its superclass (there’s only one superclass in java since there is only single inheritance) and the names of all the interfaces implemented by the class.

```
Class superclass = type.getSuperclass();
System.out.println(superclass.getName());
Class[] interfaces = type.getInterfaces();
for (int i = 0; i < interfaces.length; i++)
    System.out.println(interfaces[i].getName());
```

This type of flexibility is useful for obtaining type information at run-time, but it is not as useful as it could be, since you (the programmer) still have to ultimately know what type you are dealing with since you will need to eventually **downcast** an Object reference to a reference to the real type in order to use the type. The Object class mostly provides a convenient handle that allows you to treat all objects generically, for example, in a container type such as a Vector, Stack, Queue, etc.

More on the class `java.lang.Class`

An interesting feature of the `Class` class is a static public method call `Class.forName` that allows you to construct a `Class` object given a `String`, then ask the class object to construct an object that is an instance of the class name given in the string. This is useful for constructing objects from user input. For example:

```
public static void main(String[] args)
{
    Object[] objects = new Object[args.length];
    try {
        for(int i = 0; i < args.length; i++) {
            Class type = Class.forName(args[i]); // construct Class using args
            objects[i] = type.newInstance();
        }
    }
    catch (Exception e) { System.err.println(e); System.exit(-1); }
}
```

Of course, you have to eventually cast each constructed object down to the type of the `Class`, which you need to know in advance when you write the program. So, Java provides some dynamic type programming features, but the strong typing isn't gone.

Exceptions arising from previous example

The types of exceptions that can arise are:

1. **ClassNotFoundException** thrown by `Class.forName` because the String given does not match a known class.
2. **NoSuchMethodError** thrown by `Class.newInstance` because the Class does not define a constructor that takes no arguments. The `Class.newInstance` method constructs an object using the default constructor.
3. **IllegalAccessException** thrown by `Class.newInstance` because the default constructor is not accessible.
4. **InstantiationException** is thrown by `Class.newInstance` because the class is abstract or is an interface.