**一、A Modern Language**

**1、Enter Java:**

(1) truly distributed； can be extended and upgraded seamlessly；

Java is a modern language that addresses all three of these fronts: portability, speed, and security

(2) A virtual machine:

Java is both a compiled and an interpreted language; Java source code is turned into simple binary instructions, much like ordinary microprocessor machine code.C or C++ source is reduced to native instructions for a particular model of processor;Java source is compiled into a universal format—instructions for a *virtual machine*.Compiled Java *bytecode* is executed by a Java runtime interpreter.It executes a stack-based instruction set.

Most importantly, it does all this in accordance with a strictly defined open specification that can be implemented by anyone who wants to produce a Java-compliant virtual machine.The virtual machine and language definition provide a complete specification.For example, Java specifies the sizes and mathematical properties of all its primitive data types rather than leaving it up to the platform implementation.Java code is implicitly portable. The same Java application bytecode can run on any platform that provides a Java runtime environment.You don’t have to produce alternative versions of your application for different platforms, and you don’t have to distribute source code to end users.

Class: Classes can be maintained discretely and stored in files or archives locally or on a network server; In addition to the platform-specific runtime system, Java has a number of fundamental classes that contain architecture-dependent methods. These native methods serve as the gateway between the Java virtual machine and the real world.They are implemented in a natively compiled language on the host platform and provide low-level access to resources such as the network, the windowing system, and the host filesystem. The vast majority of Java, however, is written in Java itself—bootstrapped from these basic primitives—and is therefore portable(such as Java compiler, networking, and GUI libraries).

**JIT(just-in-time)**: In a additon to compliing source code down to portable bytecode, java has also been carefully designed so that software implementations of the runtime system can further optimize performance by compling bytecode to native machine code on the fly.(also called dynamic compilation)-->with JIT compilation, java code can execute as fast as native code and maintain its transportability and security.

There is only one intrinsic performance penalty that compiled Java code suffers at runtime for the sake of security and virtual machine design—array bounds checking.

Adaptive compilation also allows the Java runtime to take advantage of new kinds of optimizations that simply can’t be done in a statically compiled language, hence the claim that Java code can run faster than C/ C++ in some cases.

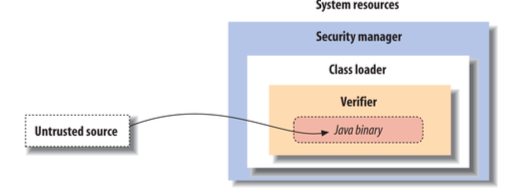
EG: Hotspot, starts out as a normal Java bytecode interpreter, but with a difference: it measures (profiles) the code as it is executing to see what parts are being executed repeatedly. Once it knows which parts of the code are crucial to performance, HotSpot compiles those sections into optimal native machine code.

Interpreter;

Compiler;

**2.Safty of Design**

**3.Safety of implementation**



Java’s first line of defense is the *bytecode verifier*. The verifier reads bytecode before it is run and makes sure it is well behaved and obeys the basic rules of the Java language.

**Class Loaders:** When a loaded class references another class name, the location of the new class is provided by the original class loader. This means that classes retrieved from a specific source can be restricted to interact only with other classes retrieved from that same location.

The search for classes always begins with the built-in Java system classes. These classes are loaded from the locations specified by the Java interpreter’s *classpath* (see Chap‐ ter 3). Classes in the classpath are loaded by the system only once and can’t be replaced. This means that it’s impossible for an application to replace fundamental system classes with its own versions that change their functionality.

Security managers: responsible for application-level security(Once a security manager is installed, it can't be replaced).A Security manager is an object that can be installed by an application to restrict access to system resources. The security manager is consulted every time the application tries to access items such as the filesystem, network ports, exernal processes, and the windowing environment; the security manager can allow or denny the request.

Application and User-level security:Digital signatures, together with certificates, are techniques for verifying that data truly comes from the source it claims to have come from and hasn’t been modified en route.

Present java7: JDBC; RMI(Remote Method Invocation); JAVA Security; JFC(Java Fundation Classes); Java 2D; Internationalization; JNDI(Java Naming and Directory Interface); JavaMail; Java 3D; Java Media; Java servlets; Java cryptographic; JavaHelp; Enterprise JavaBeans; Jini(a distributed component technology that is designed to enable distributed computing, discovery, and rendezvous of devices ranging from software tools to hardware and household appliances.); XML/XSL; Web services(Tools for creating and deploying Java-based SOAP web services);

**二、A first application**

benefits of using an IDE:an all-in-one view of Java source code with syntax higliting, navigation help, source control, integrated documentation, building, refactoring, and deployment all at your fingertips.

**1、Configuring Eclipse and Creating a Project**

**2、HelloJava**

**classes:** classes are the fundamental building blocks of most object-oriented languages. A class is a group of data items with associate functions that can perform operations on that data.

variables/feilds: the data items in a class;

methods:functions

The primary benefits of an object-oriented language are this association between data and functionality in class units and also the ability of classes to encapsulate or hide details, freeing the developer from worrying about low-level details.

**The Main method:** is the entry point for applications.

**Classes and Objects:**

**Variables and Class Types:** public static void main(String[] args): Java uses the args parameter to pass any command-line arguments supplied to the Java virtual machine (VM) into your application.

**HelloComponent:**

**Inheritance:** parent/child-->superclass/subclass ; Every class has exactly one superclass (a single parent), but possibly many subclasses. The only exception to this rule is the Object class, which sits atop the entire class hierarchy.

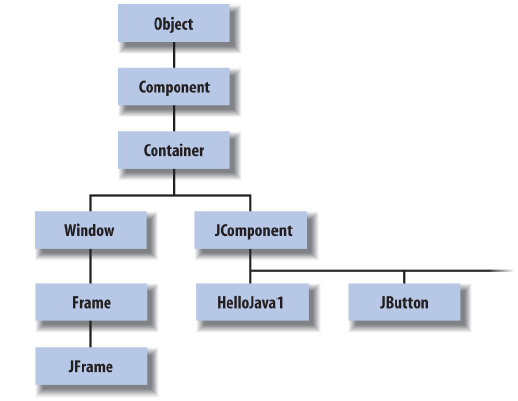
A subclass may inherit some or all the variables and methods of its superclass;Through

inheritance, the subclass can use those variables and methods as if it has declared them itself. A subclass can add variables and methods of its own, and it can also override or change the meaning of inherited methods. When we use a subclass, overridden methods are hidden(replaced) by the subclass’s own versions of them. In this way, inheritance provides a powerful mechanism whereby a subclass can refine or extend the functionality of its superclass.

polymorphism;

**The JComponent Class:**

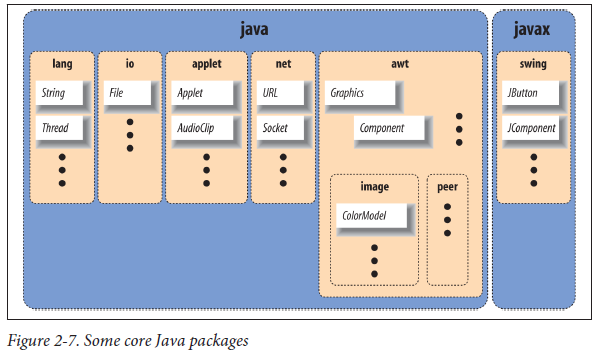
**Relationships and Finger Pointing:**



**Package and imports:**

java. is part of the core Java API and is available on any platform that supports Java;

The javax. package normally denotes a standard extension to the core platform, which may or may not be installed;



java.lang contains fundamental classes needed by the Java language itself; this package is imported automatically and that is why we didn’t need an import statement to use class names such as String or System in our examples;

The java.awt package contains classes of the older, graphical Abstract Window Toolkit;

java.net contains the networking classes;

**The paintComponent() method:**

**3、HelloJava2: The Sequel**

**Instance Variables:**

**Constructors:**A constructor is called to set up a new instance of a class;When a new object is created, Java allocates storage for it, sets instance variables to their default values, and calls the constructor method for the class to do whatever application-level setup is required.A constructor always has the same name as its class. Their sole mission in life is to configure and initialize newly born class instances, possibly using information passed to them in the parameters.

HelloComponent2 newobj = new HelloComponent2( args[0] );

Now you can pass the text on the command line when you run the application using

the following command(args[0] refers to the first command-line parameter):

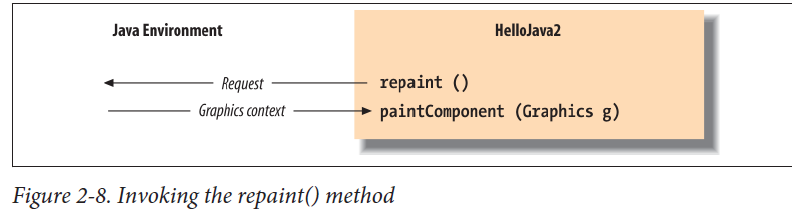
C:\> java HelloJava2 "Hello, Java!"

The special, read-only variable called this is used to explicitly refer to our object (the “current” object context).

**Events:**An event represents an action that has occurred;it contains information about the action, such as its time and location. Most events are associated with a particular GUI component in an application.

java.awt.event: provides specific Event objects that we use to get information from the user(MouseEvent, KeyEvent, and ActionEvent);

**The repaint() Method:**



**Interfaces:**

The crucial difference is that classes don’t actually inherit methods from interfaces; the interfaces merely specify the methods the class must have.

**4、HelloJava3: The button Strikes!**

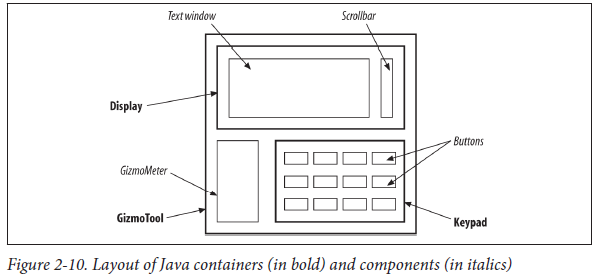
**Method Overloading:**A class can have multiple constructors, each taking different parameters and presumably using them to do different kinds of setup; When a class has multiple constructors, Java chooses the correct one based on the types of arguments used with them;

All methods in Java (not just constructors) can be overloaded; this is another aspect of the object-oriented programming principle of polymorphism;

**Components:** Component is a base class from which all of Java’s GUI components are derived; It contains variables that represent the location, shape, general appearance, and status of the object as well as methods for basic painting and event handling;

**Containers:** The Container class is an extended type of Component that maintains a list of child components and helps to group them; The Container causes its children to be displayed

and arranges them on the screen according to a particular layout strategy. Because a Container is also a Component, it can be placed alongside other Component objects in other Containers in a hierarchical fashion.



**Layout:**A LayoutManager object embodies a particular scheme for arranging components on the screen and adjusting their sizes;

**Subclassing and Subtypes:**Java allows us to use an instance of a subclass anywhere we could use an instance of its superclass;The converse, however, is not true.

**More Events and Interfaces:**

== and equals(): == is a test for identity, not equality

**Color Commentary:**

**Static Members:** class can contain variables and methods that are shared among all instances of the class. These shared members are called static variables and static methods. The most common use of static variables in a class is to hold predefined constants or unchanging objects that all the instances can use.

two advantages: (1)static values are shared by all instances of the class, the same value can be seen by all instances; (2)static members can be accessed even if no instances of the class exist.

**Arrays:** In Java, arrays are first-class objects. This means that an array itself is a type of object—one that knows how to hold an indexed list of some other type of object. when you index an array, the resulting value is an object reference—that is, a reference to the object that is located in the array’s specified slot.

{}: curly brace construct.

, , , ,:comma-separated(list of elements).

**Our Color Methods:**

**5、HelloJava4: Netscape's Revenge**

**Threads:** A thread is a separate flow of control within a program. Conceptually, threads are similar to processes. Unlike processes, multiple threads share the same program space, which means that they can share variables and methods (but also have their own local variables). Threads are also quite lightweight in comparison to processes, so it’s conceivable for a single application to be running many (perhaps hundreds or thousands) of threads concurrently.

Multithreading provides a way for an application to handle many different tasks at the same time.

The difficulty lies in making sure routines are implemented so they can be run concurrently by more than one thread at a time.

**The Thread Class:** We call the Thread’s start() method to begin execution. Once the thread starts, it continues to run until it completes its work, we interrupt it, or we stop the application.

It’s not usually desirable to create a subclass of Thread to contain our run() method.

The Thread class has a constructor that takes an object as its argument. If we create a

Thread object using this constructor and call its start() method, the Thread executes

the run() method of the argument object rather than its own. In order to accomplish

this, Java needs a guarantee that the object we are passing it does indeed contain a

compatible run() method. We already know how to make such a guarantee: we use an

interface. Java provides an interface named Runnable that must be implemented by any

class that wants to become a Thread.

（1）定义自己的实现了Runnable接口的类

   MyRunnable implements Runnable

   {

       public void run()

        {

           your code;

        }

   }

   MyRunnable myrunnable = new MyRunnable();

   Thread t = new Thread(myrunnable);

   t.start;

   （2）直接定义Thread类的子类(不建议这样使用)

   class MyThread extends Thread

   {

       public void run()

        {

            your code;

        }

   }

   MyThread t = new MyThread();

   t.start;

   采用继承Thread类方式：

     优点：编写简单，如果需要访问当前线程，无需使用Thread.currentThread()方法，直接使用this，即可获得当前线程。

     缺点：因为线程类已经继承了Thread类，所以不能再继承其他的父类。

   采用实现Runnable接口方式：

     优点：线程类只是实现了Runable接口，还可以继承其他的类。在这种方式下，可以多个线程共享同一个目标对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将CPU代码和数据分开，形成清晰的模型，较好地体现了面向对象的思想。

     缺点：编程稍微复杂，如果需要访问当前线程，必须使用Thread.currentThread()方法。

     不要直接调用Thread或Runnable的Run方法，那样只是在当前线程中执行任务并不会创建新的任务，用start方法才会创建新的任务。

    尽量从机制上避免这种并行任务，因为代价太大，可以只用线程池。

**The Runnable Interface:**

**Starting the Thread**

**Running Code in the Thread:**

?a:b C language-style ternary operator

**Exceptions:** An exception is a message that is sent, normally in response to an error, during the execution of a statement or a method.When an exceptional condition arises, an object is created that contains information about the particular problem or condition.Java stops execution at the place where the exception occurred, and the exception object is said to be thrown by that section of code.The try/catch construct allows you to catch exceptions for a section of code. If an exception is caused by any statement inside a try clause, Java attempts to deliver the exception to the appropriate catch clause. A catch clause looks like a method declaration with one argument and no return type. Depending on the type of exception thrown, the appropriate catch clause is executed.

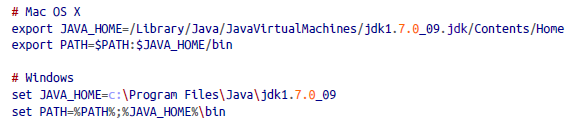
If there is no try/catch clause surrounding the code, or a matching catch clause is not found, the exception is thrown up to the calling method. If the exception is not caught there, it’s thrown up to another level, and so on until the exception is handled or the Java VM prints an error and exits. This provides a very flexible error-handling mechanism so that exceptions in deeply nested calls can bubble up to the surface of the call stack for handling.

Under some circumstances, Thread’s sleep() method can throw an InterruptedException, indicating that it was interrupted by another thread.

**Synchronization:** The synchronized modifier tells Java to acquire a lock for the object that contains the method before executing that method. Only one method in the object can have the lock at any given time, which means that only one synchronized method in that object can be running at a time.When the method is done, it releases the lock on the class. The synchronized keyword in Java provides locking at the language level. This means there is no way that you can forget to unlock a class. Even if the method throws an exception or the thread is terminated, Java will release the lock.

**三、Tools of the Trade**

**1、JDK Environment**



java -version

javac -version

**2、The Java VM**

The Java VM performs all the runtime activities of Java. It loads Java class files, verifies classes from untrusted sources, and executes the compiled bytecode. It manages memory and system resources. Good implementations also perform dynamic optimization, compiling Java bytecode into native machine instructions.

**3、Running Java Applications**

To run the application, start the VM, specifying that class as an argument. You can also specify options to the interpreter as well as arguments to be passed to the application:

java [interpreter options] class\_name [program arguments]

The class should be specified as a fully qualified class name, including the package name, if any. Note, however, that you don’t include the .class file extension.

clipboard.png

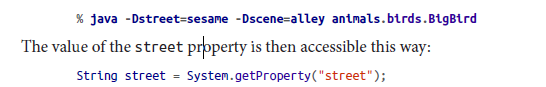
The classpath can be specified either by an environment variable or with the commandline option -classpath. If both are present, the command-line option is used. Alternately, the java command can be used to launch an “executable” Java archive (JAR) file:

java -jar spaceblaster.jar

public static void main ( String [] myArgs )

The fact that main() is a public and static method simply means that it is globally accessible and that it can be called directly by name.The main() method’s single argument, the array of String objects, holds the commandline arguments passed to the application.

Java allows any number of system property values to be passed to the application when the VM is started. System properties are simply name-value string pairs that are available to the application through the static System.getProperty() method. Each system property is passed to the interpreter on the command line using the -D option followed by name=value



**4、The Classpath**

The PATH environment variable is a colon-separated list of directories that are searched, in order, when the user types the name of a command. The Java CLASSPATH environment variable, similarly, is a list of locations that are searched for Java class files. An element of the classpath can be a directory or a JAR file. Java also supports archives in the conventional ZIP format, but JAR and ZIP are really the same format.

For the Java runtime, java, and the Java compiler, javac, the classpath can also be specified

with the -classpath option:

javac -classpath /home/pat/classes:/utils/utils.jar:. Foo.java

If you don’t specify the CLASSPATH environment variable or command-line option, the classpath defaults to the current directory (.); this means that the files in your current directory are normally available.

**javap:** With javap, you can print a description of a compiled class.

javap java.util.Stack

**5、The Java Compiler**

**javac:** The javac compiler is written entirely in Java, so it’s available for any platform that supports the Java runtime system. javac turns Java source code into a compiled class that contains Java bytecode.

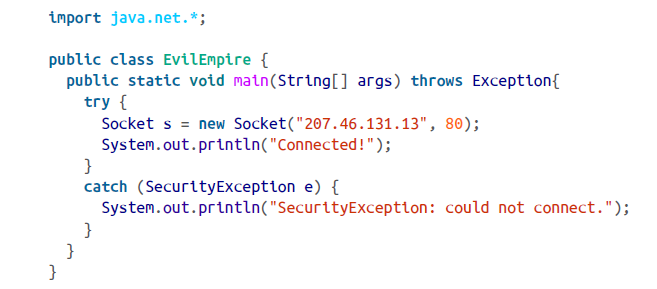
javac allows one public class per file and insists that the file have the same name as the class. If the filename and class name don’t match, javac issues a compilation error. A single file can contain multiple classes, as long as only one of the classes is public and is named for the file.

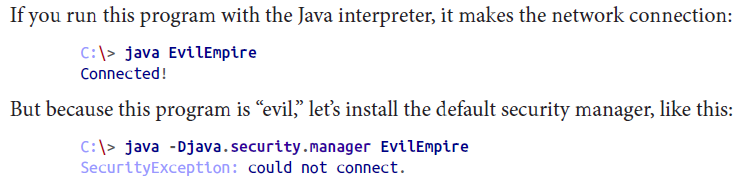
You can use the -d option with javac to specify an alternative directory for storing the class files javac generates. The specified directory is used as the root of the class hierarchy.

By default, however, javac checks only source files that are referenced directly from other source files. This means that if you have an out-of-date class file that is referenced only by an up-to-date class file, it may not be noticed and recompiled.----most projects use a real build utility such as Apache’s Ant to manage builds, packaging, and more.

**6、JAR Files**

**7、Policy Files**



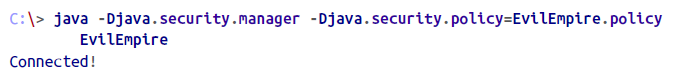


To permit our EvilEmpire example to make a network connection, we need to create a

policy file that contains the appropriate permission. A handy utility called policytool,

included with the JDK, helps make policy files. Fire it up from a command line like this:

clipboard.png



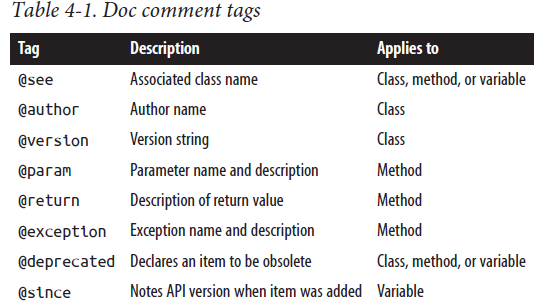
**四、The Java Language**

**1、Text Encoding**

Java supports internationalization is through the Unicode character set（Unicode is a worldwide standard that supports the scripts of most languages）；uses at least two bytes to represent each symbol internally；

Java source code can be written using Unicode and stored in any number of character encodings, ranging from a full binary form to ASCII-encoded Unicode character values

**2、Comments**



**3、Types**

In a statically typed language, such as C or C++, the type of a data element is a simple, unchanging attribute that often corresponds directly to some underlying hardware phenomenon, such as a register or a pointer value;In a more dynamic language such as Smalltalk or Lisp, variables can be assigned arbitrary elements and can effectively change their type throughout their lifetime;Scripting languages such as Perl achieve ease of use by providing drastically simplified type systems in which only certain data elements can be stored in variables, and values are unified into a common representation, such as strings;

Java data types fall into two categories. Primitive types represent simple values that have

built-in functionality in the language; they are fixed elements, such as literal constants

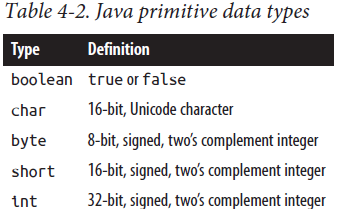
and numbers. Reference types (or class types) include objects and arrays; they are called

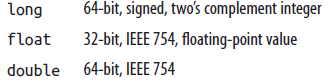
reference types because they “refer to” a large data type that is passed “by reference,” as

we’ll explain shortly. Generic types are really just a kind of composition (combination)

of class types and are therefore reference types as well.

(1)Primitive types: Numbers, characters, and Boolean values are fundamental elements in Java;





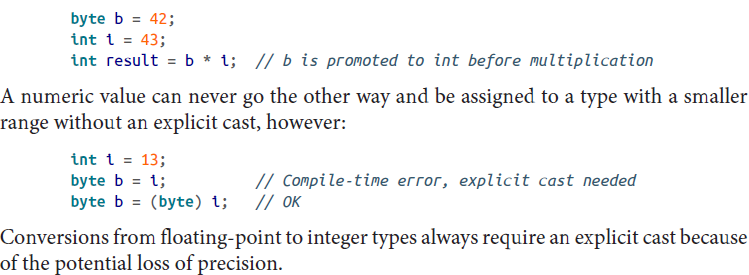
strictfp( class modifier)

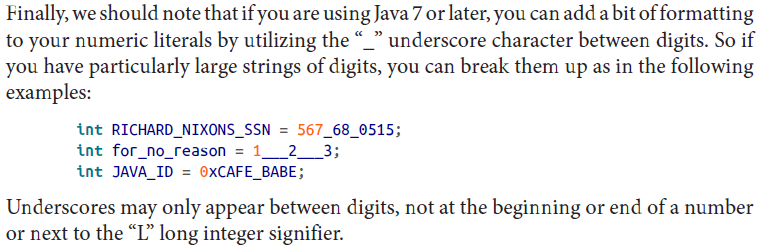
octal:8

decimal:10

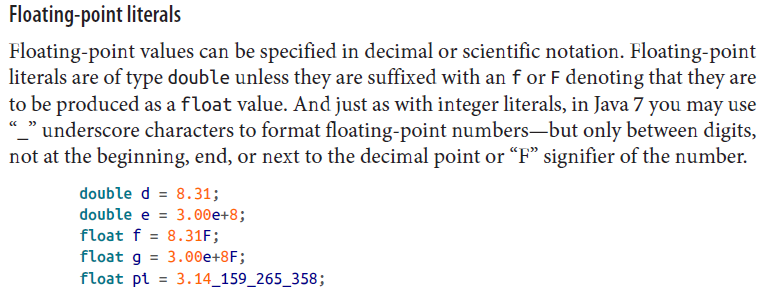
hexadecimal:16

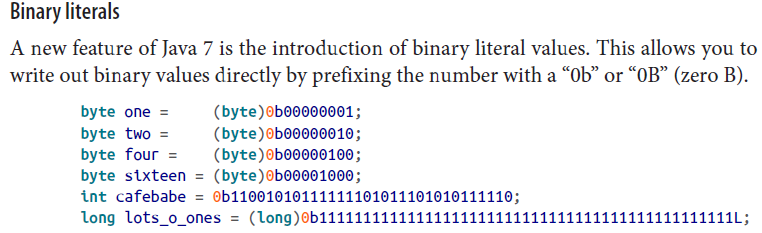
arithmetic promotion: When a numeric type is used in an assignment or an expression involving a “larger” type with a greater range, it can be promoted to the bigger type.For example, when multiplying a byte value by an int value, the compiler promotes the byte to an int first:

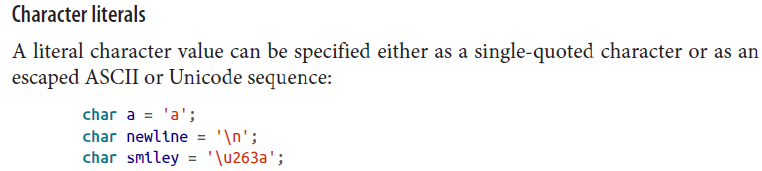




the '\_' will be ignored, just used to improve the readability of your code.







(2)Reference Types

In an object-oriented language like Java, you create new, complex data types from simple primitives by creating a class. Each class then serves as a new type in the language.

Primitive types in Java are used and passed “by value.” In other words, when a primitive

value like an int is assigned to a variable or passed as an argument to a method, it’s simply copied. Reference types (class types), on the other hand, are always accessed “by reference.” A reference is simply a handle or a name for an object. What a variable of a reference type holds is a “pointer” to an object of its type (or of a subtype, as described earlier). When the reference is assigned to a variable or passed to a method, only the reference is copied, not the object to which it’s pointing.

the argument passed to a method (a local variable from the method’s point of view) is actually a third reference to the Foo object, in addition to myFoo and anotherFoo. The method can alter the state of the Foo object through that reference (calling its methods or altering its variables), but it can’t change the caller’s notion of the reference to myFoo: that is, the method can’t change the caller’s myFoo to point to a different Foo object; it can change only its own reference.

(3)A Word About Strings

Strings in java are objects; they are therefore a reference type;

+,+=:the only two overloaded operator in Java;

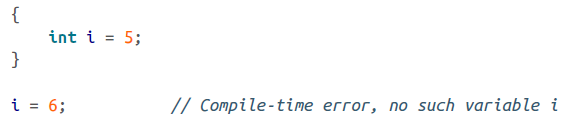
**4、Statements and Expressions**

Variable declarations and assignments, such as those in the previous section,are statements, as are basic language structures such as if/then conditionals and loops.

curly brace:

open curly brace: { close curly brace:}

Variable declarations are limited in scope to their enclosing code block—that is, they can’t be seen outside of the nearest set of braces:



**;** multiple comma-separated expressions

the enhanced for loop:

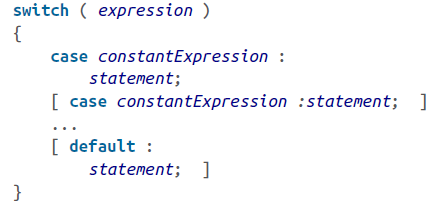
clipboard.png

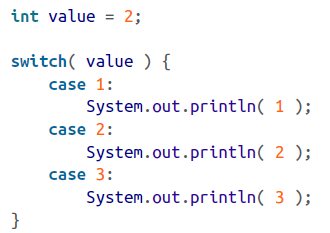
**Switch Statements:**

**integer, numberic type, string type, enum type**

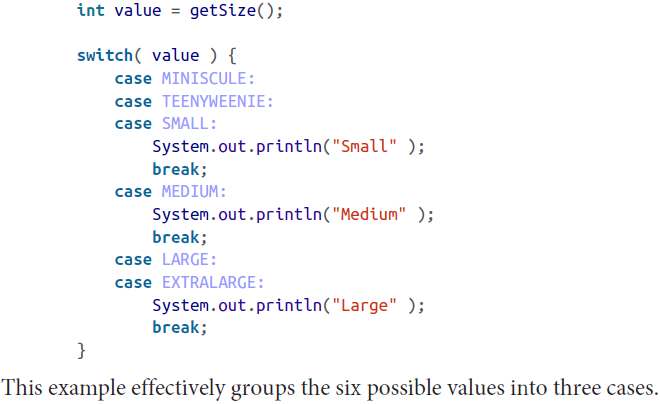
The most common form of the Java switch statement takes an integer (or a numeric type argument that can be automatically “promoted” to an integer type), a string type

argument, or an “enum” type (discussed shortly) and selects among a number of alternative, constant case branches:





prints 2, 3! should add "break" to each branch;



page

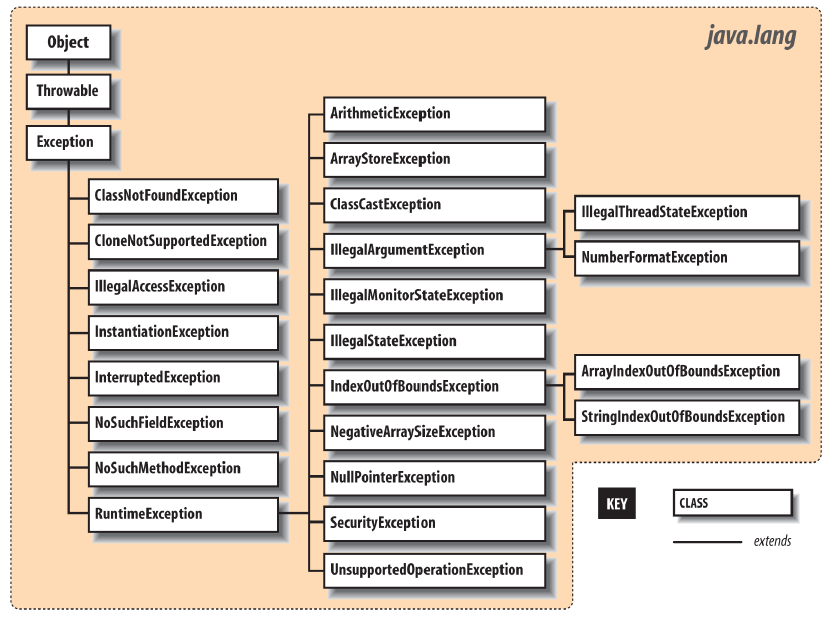
**5、Exceptions**

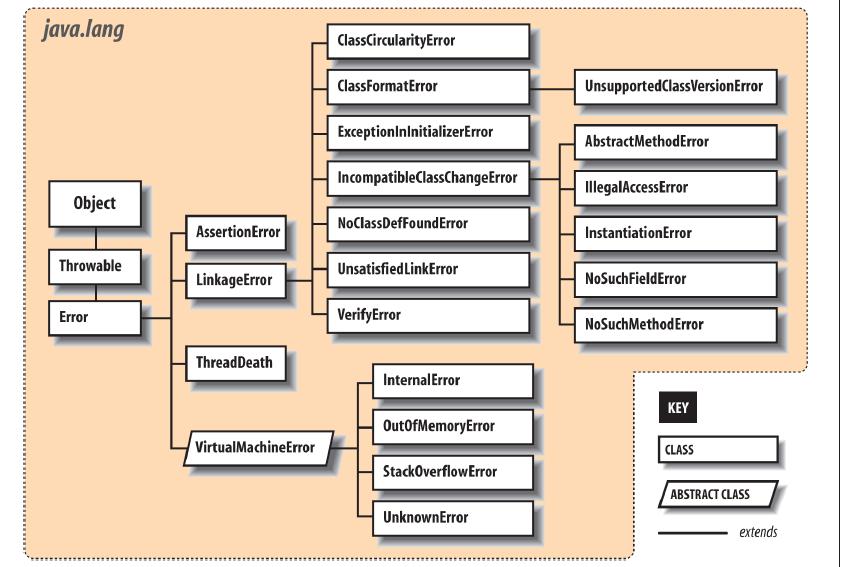
**(1)Exceptions and Error Classes:**

<1>root(interface): java.lang.Throwable

class root: java.lang.Exception

class root: java.lang.Error





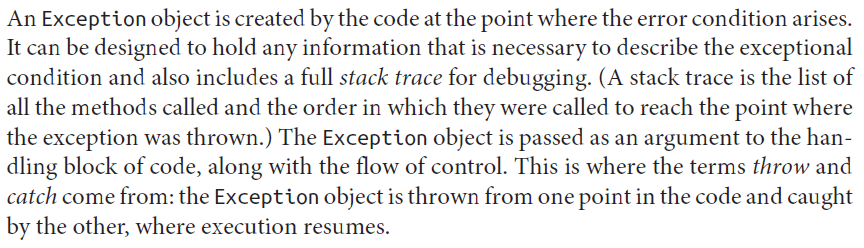
<2>difference between Error and Exception:

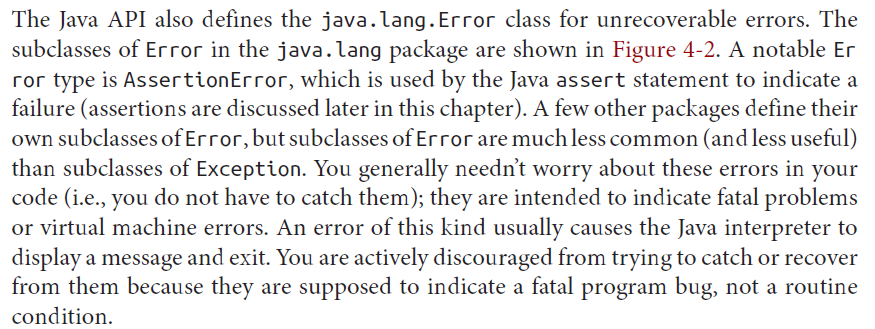
Error: An Error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions.

Exception: The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch.

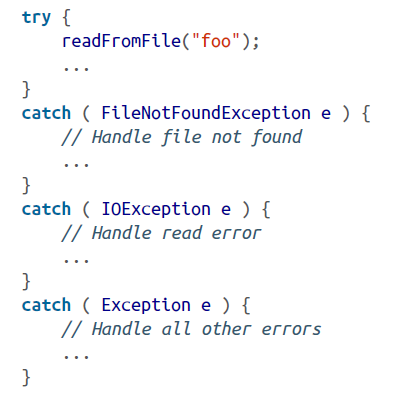
unchecked exceptions: Error and RuntimeException (eg: ArrayIndexOutofBoundException,ClassCastException), don't try to catch them, try to modify the code!

checked exceptions: all other exceptions exception RuntimeExcepions(eg: FileNotFoundException, ParseException..), try to catch them

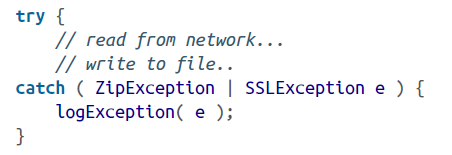




<3>Exception Handling



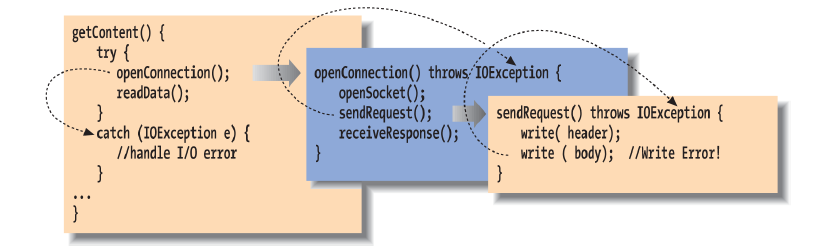
The catch clauses are evaluated in order, and the first assignable match is taken. At most, one catch clause is executed, which means that the exceptions should be listed from most to least specific. If an earlier statement fails,execution jumps immediately to the catch clauses; later statements are never executed.



In Java 7, there's an alternative to using multiple catch clauses, and that is to handle multiple discrete exception types in a single catch clause using the "|". We receive both types of exceptions in the same catch clause. The actual type of the e variable is the closest parent class type to which they are both assignable.

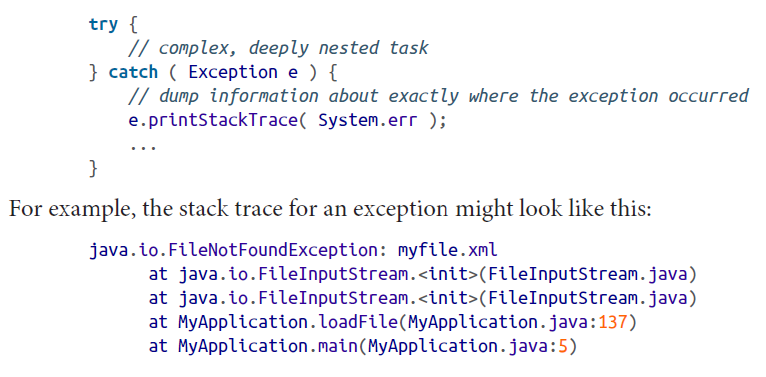
<4>Bubbling Up

If there's no enclosing try/catch statement, the exception pops up from the method in which it originated and is thrown from that method to its caller. If that point in the calling method is within a try clause, control passes to the corresponding catch clause. Otherwise, the exception continues propagating up the call stack, from one method to its caller. In this way, the exception bubbles up until it's caught, or until it pops out of the top of the program, terminating it with a runtime error message.



<5>Stack Traces

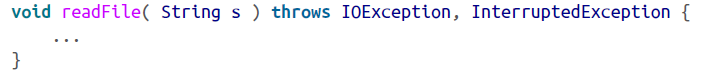
Element zero of the array is the top of the stack, the final line of code that caused the exception; subsequent elements step back one method call each until the original main() method is reached.



<6>Checked and Unchecked Exceptions

Most application-level exceptions are checked, which means that any method that throws one, either by generating it itself (as we’ll discuss later) or by ignoring one that occurs within it, must declare that it can throw that type of exception in a special throws clause in its method declaration. Methods have to declare the checked exceptions they can throw or allow to be thrown.

If we had to throw multiple types of exceptions, we could declare them separated by commas.



Unchecked: exceptions that are subclasses of either the class java.lang.RuntimeException or the class java.lang.Error.

Checked exceptions are intended to cover application-level problems, such as missing files and unavailable hosts. Unchecked exceptions are intended for system-level problems, such as “out of memory” and “array index out of bounds.” While these may indicate application-level programming errors, they can occur almost anywhere and usually aren’t possible to recover from.

To sum up, checked exceptions are problems that a reasonable application should try to handle gracefully; unchecked exceptions (runtime exceptions or errors) are problems from which we would not normally expect our software to recover. Error types are those explicitly intended to be conditions that we should not normally try to handle or recover from.

<7>Throwing Exceptions

throw new IOException();

throw new IOException("Sunspots!");

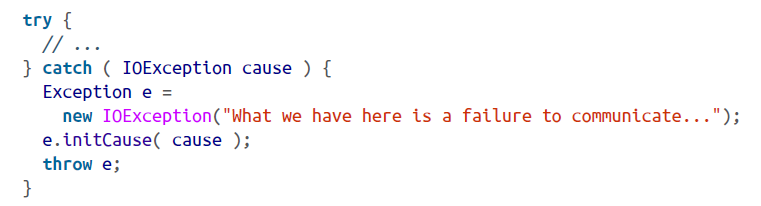
(You can retrieve this string by using the Exception object’s getMessage() method. Often, though, you can just print (or toString()) the exception object itself to get the message and stack trace.)

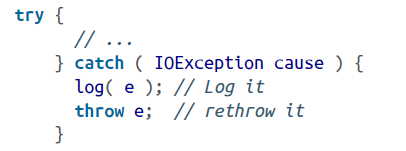
<8>Chaining and rethrowing exceptions

Sometimes you’ll want to take some action based on an exception and then turn around and throw a new exception in its place. This is common when building frameworks where low-level detailed exceptions are handled and represented by higher-level exceptions that can be managed more easily. For example, you might want to catch an IOException in a communications package, possibly perform some cleanup, and ultimately throw a higher-level exception of your own, maybe something like LostServerConnection. The base Exception class can be constructed with an exception as an argument or the standard String message and an exception:

throw new Exception( "Here's the story...", causalException );

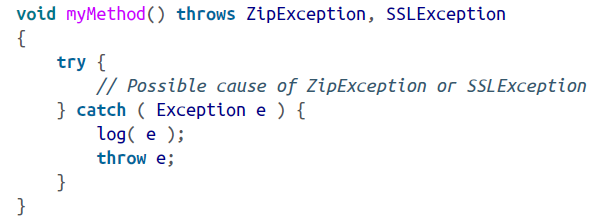
You can get access to the wrapped exception later with the getCause() method. More importantly, Java automatically prints both exceptions and their respective stack traces if you print the exception or if it is shown to the user.





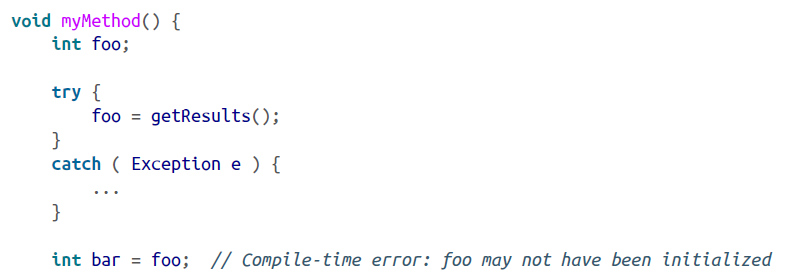
<9>Narrowed rethrow

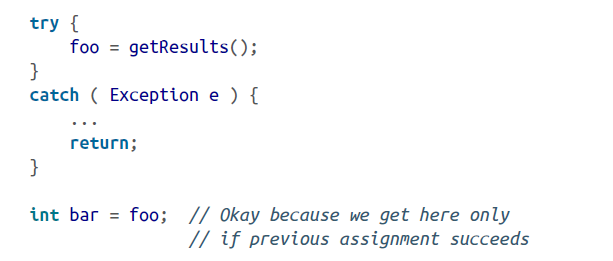
Prior to Java 7 if you wanted to handle a bunch of exception types in a single catch clause and then rethrow the original exception, you would inevitably end up widening the declared exception type to what was required to catch them all or having to do a lot of work to avoid that. In Java 7, the compiler has become smarter and can now do most of the work for us by allowing us to narrow the type of exceptions thrown back to the original types in most cases.



<10>Try Creep

The try statement imposes a condition on the statements that it guards. It says that if an exception occurs within it, the remaining statements are abandoned. This has consequences for local variable initialization. If the compiler can’t determine whether a local variable assignment placed inside a try/catch block will happen.





<11>The finally Clause

Any statements in the body of the finally clause are guaranteed to be executed no matter how control leaves the try body, whether an exception was thrown or not:

try {

// Do something here

}

catch ( FileNotFoundException e ) {

...

}

catch ( IOException e ) {

...

}

catch ( Exception e ) {

...

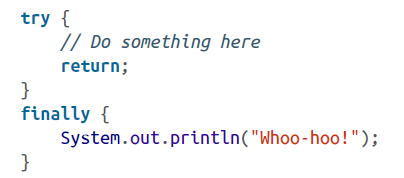
}

finally {

// Cleanup here is always executed

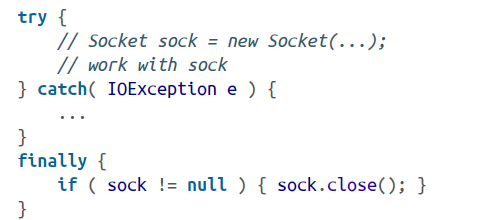
}

If the statements in the try execute cleanly, or if we perform a return , break, or continue, the statements in the finally clause are still executed. To guarantee that some operations will run, we can even use try and finally without any catch clauses:



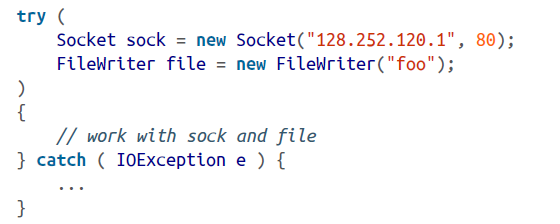
<12>Try with Rersources

A common use of the finally clause is to ensure that resources used in a try clause are cleaned up, no matter how the code exits the block.



two problems: first, it requires extra work to carry out this pattern in all of your code, including important things like null checks as shown in our example; second, if you are juggling multiple resources in a single finally block, you have the possibility of your cleanup code throwing an exception (e.g., on close()) and leaving the job unfinished.

In Java 7, things have been greatly simplified via the new “try with resources” form of the try clause. In this form, you may place one or more resource initialization statements within parentheses after a try keyword and those resources will automatically be “closed” for you when control leaves the try block.



In this example, we initialize both a Socket object and a FileWriter object within the try-with-resources clause and use them within the body of the try statement. When control leaves the try statement, either after successful completion or via an exception, both resources are automatically closed by calling their close() method. Resources are closed in the reverse of the order in which they were constructed, so dependencies among them can be accommodated. This behavior is supported for any class that implements the AutoCloseable interface.

All the exceptions are merely “suppressed” and can be retrieved via the Throwable getSuppressed() method of the exception thrown to the caller. This returns an array of all of the supressed exceptions.

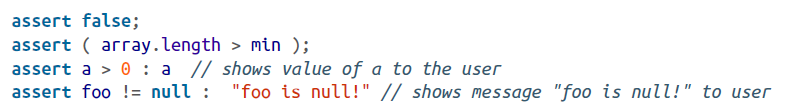
<12>Performance Issues

Guarding against an exception being thrown (using a try) is free; However, throwing an exception is not free.

The result is that you should throw exceptions only in truly “exceptional” circumstances and avoid using them for expected conditions.The general rule should be that exceptions are used for “out of bounds” or abnormal situations, not routine and expected conditions.

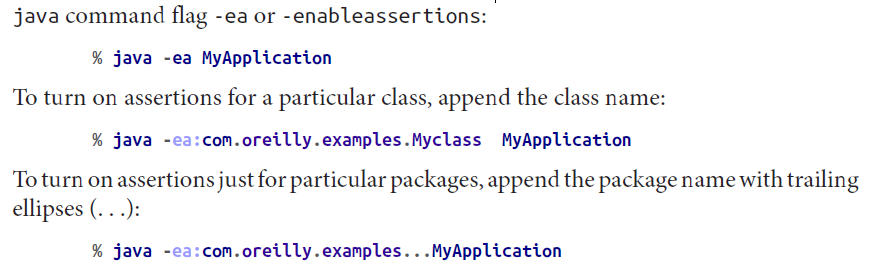
**6、Assertions**

Assertions are distinct from other kinds of tests because they check conditions that should never be violated at a logical level: if the assertion fails, the application is to be considered broken and generally halts with an appropriate error message. Assertions are supported directly by the Java language and they can be turned on or off at runtime to remove any performance penalty of including them in your code.



<1>Enabling and Disabling Assertions:

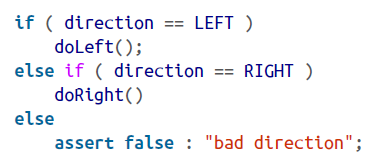
You can enable and disable assertions for an entire application or on a package-by-package or even class-by-class basis. By default, assertions are turned off in Java.

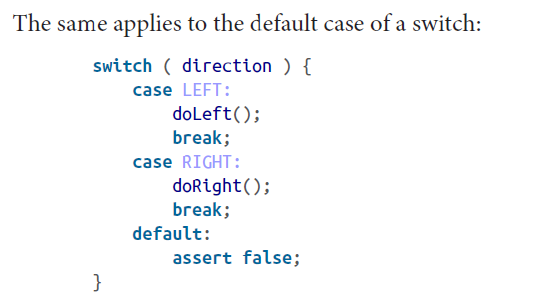


You can combine all this to achieve arbitrary groupings like this:

% java -ea:com.oreilly.examples... -da:com.oreilly.examples.text-ea:com.oreilly.examples.text.MonkeyTypewriters MyApplication

<2>Using Assertions





**7、Arrays**

<1>

base type of the array: the type of the elements of the array.

The number of elements it holds is a fixed attribute called its length. supports arrays of all primitive and reference types.

Array objects differ from other objects in Java in three respects:

Java implicitly creates a special array class type for us whenever we declare a new type of array;

Java lets us use the [] operator to access array elements so that arrays look as we expect. We could implement our own classes that act like arrays, but we would have to settle for having methods such as get() and set() instead of using the special [] notation;

Java provides a corresponding special form of the new operator that lets us construct an instance of an array with a specified length with the [] notation or initialize it directly from a structured list of values.

**Note: () parenthese (parenthesis);**

**[]square brackets**

**[][] multiple bracket pairs**

**<>angle brackets**

**{}braces; or curly-brace**

**1,2,3,... comma-separated**

<2>Array Types

int [] arrayOfInts; // preferred

int arrayOfInts []; // C-style

String [] someStrings;

Button [] someButtons;

<3>Array Creation and Initialization

The new operator is used to create an instance of an array. After the new operator, we specify the base type of the array and its length with a bracketed integer expression:

arrayOfInts = new int [42];

someStrings = new String [ number + 2 ];

double [] someNumbers = new double [20];

Component [] widgets = new Component [12];

int [] primes = { 2, 3, 5, 7, 7+4 }; // C style initializing

After creation, the array elements are initialized to the default values for their type. For numeric types, this means the elements are initially zero.

The elements of an array of objects are references to the objects—just like individual variables they point to—but do not actually contain instances of the objects. The default value of each element is therefore null until we assign instances of appropriate objects.In many other languages, the act of creating an array is the same as allocating storage for its elements. In Java, a newly allocated array of objects actually contains only reference variables, each with the value null. That’s not to say that there is no memory associated with an empty array; memory is needed to hold those references.

<4>Using Arrays

String [] musketeers = { "one", "two", "three" };

int num = musketeers.length; // num == 3

length is the only accessible field of an array; it is a variable, not a method;

enhanced for loop:

for (Button b : keyPad)

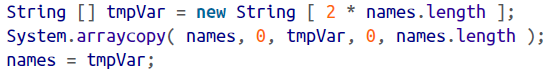
System.out.println(b);

Attempting to access an element that is outside the range of the array generates an ArrayIndexOutOfBoundsException. This is a type of RuntimeException, so you can either catch and handle it yourself if you really expect it, or ignore it.

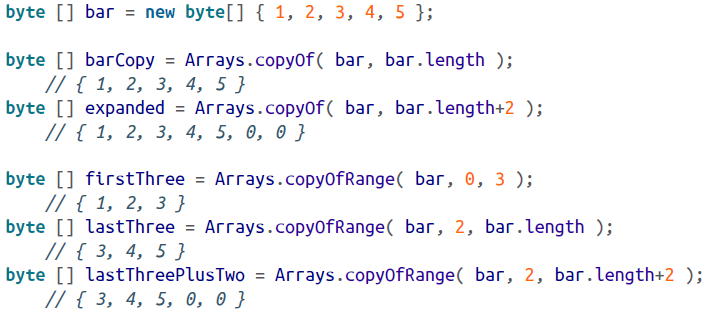
It’s a common task to copy a range of elements from one array into another. One way to copy arrays is to use the low-level arraycopy() method of the System class:

System.arraycopy( source, sourceStart, destination, destStart, length );

The following example doubles the size of the names array from an earlier example:



An easier way is to use the java.util.ArrayscopyOf() and copyOfRange() methods:



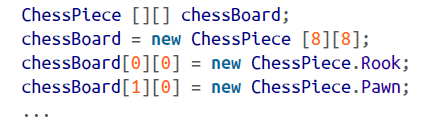
<5>Anonymous Arrays

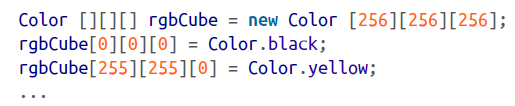
Often it is convenient to create “throwaway” arrays, arrays that are used in one place and never referenced anywhere else. Such arrays don’t need a name because you never need to refer to them again in that context. Let’s say you need to call a method named setPets(), which takes an array of Animal objects as arguments. Provided Cat and Dog are subclasses of Animal, here’s how to call setPets() using an anonymous array:



<6>Multidimensional Arrays

You create a multidimensional array with C-like syntax, using multiple bracket pairs, one for each dimension.





We don’t necessarily have to specify the dimension sizes of a multidimensional array with a single new operation. The syntax of the new operator lets us leave the sizes of some dimensions unspecified. The size of at least the first dimension (the most significant dimension of the array) has to be specified, but the sizes of any number of trailing, less significant array dimensions may be left undefined.

boolean [][] checkerBoard;

checkerBoard = new boolean [8][];

checkerBoard[0] = new boolean [8];

checkerBoard[1] = new boolean [8];

...

checkerBoard[7] = new boolean [8];

equivalant to:

boolean [][] checkerBoard = new boolean [8][8];

<7>Inside Arrays

Arrays are instances of special array classes in the Java language.........

**五、Objects in Java**

general design guidelines：

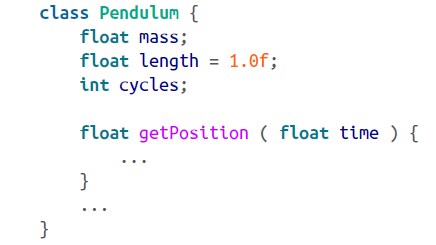
Hide as much of your implementation as possible.

Specialize objects only when you have to—use composition instead of inheritance.

Minimize relationships between objects and try to organize related objects in packages.

**1.Classses**

Classes are the building blocks of a Java application. A class can contain methods (functions), variables, initialization code and other classes.It serves as a blueprint for making class instances.



**a.** Two kinds of variables can be defined in a class: instance variables and static variables.

Every object instance has its own set of instance variables; the values of these variables in one instance of an object can differ from the values in another object. Static variables, in contrast, are shared among all instances of an object. In either case, if you don’t initialize a variable when you declare it, it’s given a default value appropriate for its type (null, zero, or false).

**b.** As with variables,methods defined in a class may be instance methods or static methods.

An instance method is associated with just one instance of the class, but the relationship isn’t quite as simple as it is for variables. Instance methods are accessed through an object instance, but the object doesn’t really have its own “copy” of the methods (there is no duplication of code). Instead, the association means that instance methods can “see” and operate on the values of the instance variables of the object. Static methods are really more like global functions, as they are associated with a class by name only.

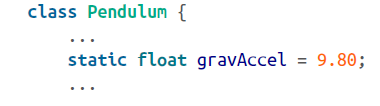
**<1> Accessing Fields and Methods**

Inside a class, we can access variables and call methods of the class directly by name.

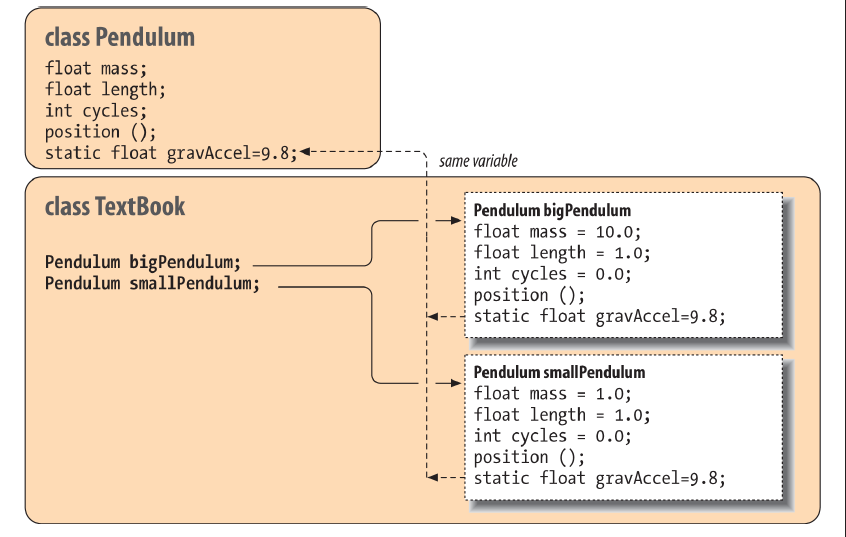
Other classes access members of an object through a reference, using the dot selector notation.

**<2>Static Members**

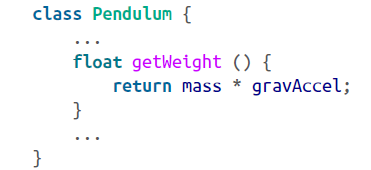
Members that are declared with the static modifier live in the class and are shared by all instances of the class. Variables declared with the static modifier are called static variables or class variables; similarly, these kinds of methods are called static methods or class methods.



That means that it is associated with the class, not with an individual instance and if we change its value (either directly or through any instance of a Pendulum), the value changes for all Pendulum objects.



Static members can be accessed like instance members. Inside our Pendulum class, we can refer to gravAccel like any other variable:

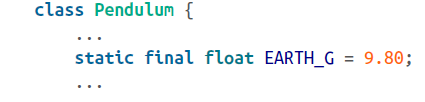


However, since static members exist in the class itself, independent of any instance, we can also access them directly through the class.

clipboard.png

This changes the value of gravAccel as seen by all instances.

It’s also common to use static variables to define constant values. In this case, we use the static modifier along with the final modifier.



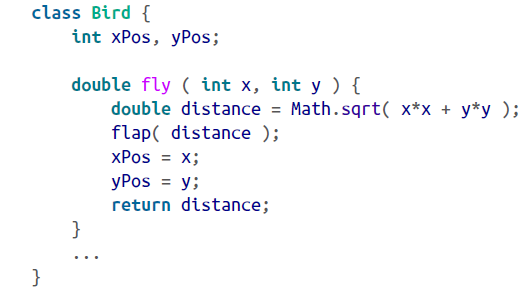
It can be accessed through the class Pendulum or its instances, but its value can’t be changed at runtime.

Constants versus enumerations: Enumerations were added to the Java language to replace this identifier(static final) usage with a mechanism that is both safer and, in some cases, more efficient.

clipboard.png

This enumeration creates not only the values, but also a new type, PendulumTypes, whose value is limited to one of the three discrete identifiers. Calling code can refer to the values as it did through our class: PendulumTypes.Simple.

**2.Methods**



Our method has a fixed number of arguments (two); however, methods can have variable-length argument lists, which allow the method to specify that it can take any number of arguments and sort them itself at runtime.

**<1>Local Variables**

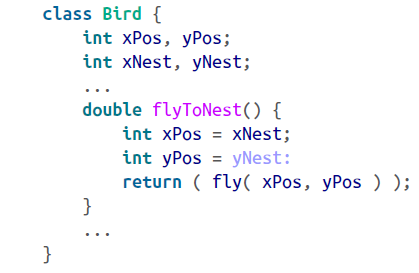
A local variable is temporary; it exists only within the scope (the block) of its method. Local variables are allocated when a method is invoked; they are normally destroyed when the method returns.

**If the method is executing concurrently in different threads, each thread has its own version of the method’s local variables**. A method’s arguments also serve as local variables within the scope of the method; the only difference is that they are initialized by being passed in from the caller of the method.

An object created within a method and assigned to a local variable may or may not persist after the method has returned. As with all objects in Java, it depends on whether any references to the object remain. If an object is created, assigned to a local variable, and never used anywhere else, that object is no longer referenced when the local variable disappears from scope, so garbage collection removes the object. If, however, we assign the object to an instance variable of an object, pass it as an argument to another method, or pass it back as a return value, it may be saved by another variable holding its reference.

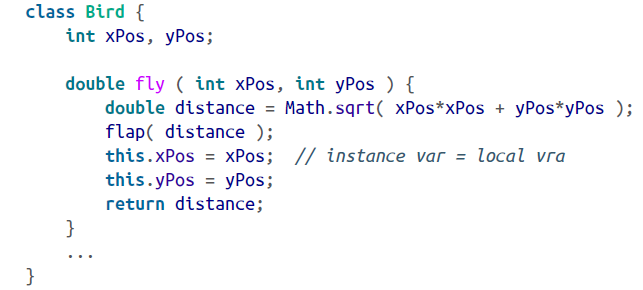
**<2>Shadowing**

In the following example, the local variables xPos and yPos hide the instance variables of the same name:



When we set the values of the local variables in flyToNest(), it has no effect on the values of the instance variables.

The "this" reference: You can use the special reference this any time you need to refer explicitly to the current object or a member of the current object.

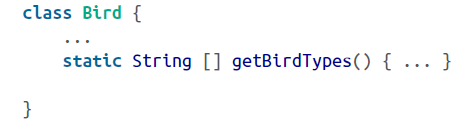


In this example, the expression this.xPos refers to the instance variable xPos and assigns it the value of the local variable xPos

**<3>Static Methods**

Static methods (class methods), like static variables, belong to the class and not to individual instances of the class.It can be invoked by name, through the class name, without any objects around. A static method can directly access only other static members (static variables and other static methods) of the class. It can’t directly see any instance variables or call any instance methods.

Static methods are particularly useful for utility methods that perform work that is useful either independently of instances or in working on instances.

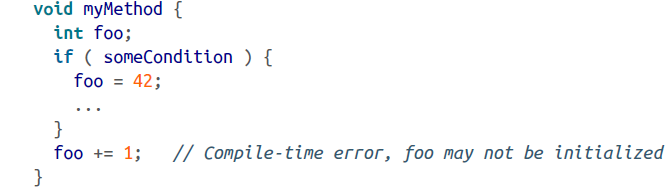


We can use getBirdTypes() from within an instance of Bird, just like an instance method. However, we can also call it from other classes, using the Bird class name: String [] names = Bird.getBirdTypes();

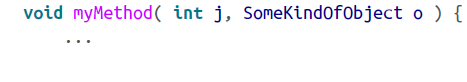
Static methods also play an important role in various design patterns, where you limit the use of the new operator for a class to one method—a static method called a factory method.

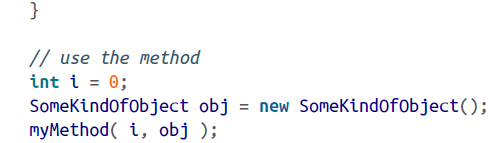
**<4>Initializing Local Variables**

Unlike instance variables, local variables must be initialized before they can be used.



**<5>Argument Passing and References**



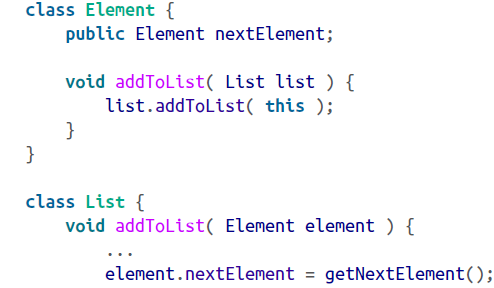


The first argument, i, is passed by value; when the method is called, the value of i is copied into the method’s parameter (a local variable to it) named j. If myMethod() changes the value of j, it’s changing only its copy of the local variable. In the same way, a copy of the reference to obj is placed into the reference variable o of myMethod(). Both references refer to the same object, so any changes made through either reference affect the actual (single) object instance. However, if myMethod() changes the reference o itself—to point to another object—it’s affecting only its local variable reference. It doesn’t affect the caller’s variable obj, which still refers to the original object. In this sense, passing the reference is like passing a pointer in C and unlike passing by reference in C++.

What if myMethod() needs to modify the calling method’s notion of the obj reference as well (i.e., make obj point to a different object)? The easy way to do that is to wrap obj inside some kind of object.

clipboard.png

Another possibility is to use this to pass a reference to the calling object. In that case, the calling object serves as the wrapper for the reference.

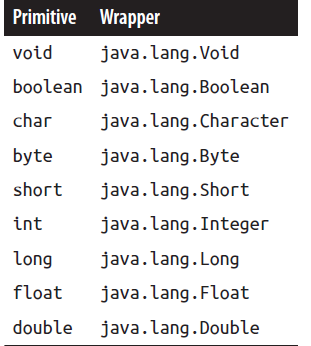


In this code, the Element class represents one element; it includes a method for adding itself to the list.

The same technique can be used in conjunction with interfaces to implement callbacks for arbitrary method invocations.

**<6>Wrappers for Primitive Types**

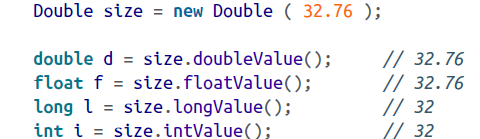
Java supplies a standard wrapper class for each of the primitive types:



You can construct a wrapper object from a primitive value or from a String representation of the value. The following statements are equivalent:

clipboard.png

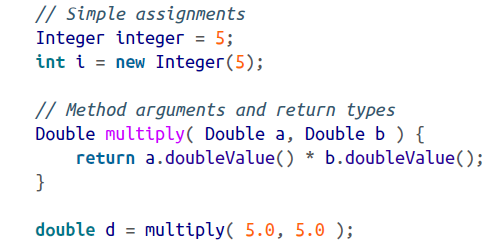
Each of the numeric type wrappers implements the java.lang.Number interface



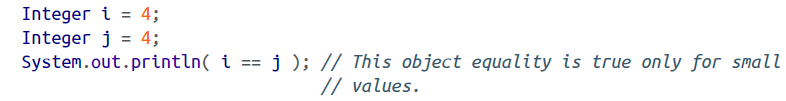
This code is equivalent to casting the primitive double value to the various types.

**<7>Autoboxing and Unboxing of Primitives**

The Java compiler automatically wraps primitives in their wrapper types and unwraps them where appropriate.

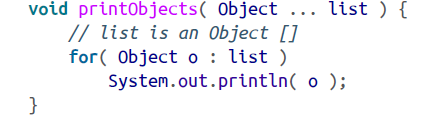


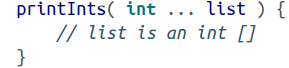
Performance: We can anticipate that allowing Java to box and unbox primitives in performance-critical areas will not be as fast as using primitives directly. One aspect of this to consider is how many new objects are being created and reclaimed by the garbage collector. While in general Java may be forced to create a new object for each boxed primitive, there are optimizations for a small range of values. Java guarantees that the Boolean values true and false, as well as “small” valued numeric types ranging from 0 to 127 for bytes and chars and from –128 to 127 for shorts and integers, are interned. Saying that they are interned means that instead of creating a new object each time, Java reuses the same object on subsequent boxings. This is safe because primitive wrappers are immutable and cannot be changed.

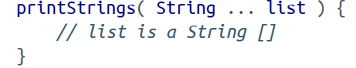


**<8>Variable-Length Argument Lists**

A method accepting a variable argument list is equivalent to a method accepting an array of some type of object. The difference is that the compiler makes the method call accept individual, comma-separated values, and then packs them into the array for us. The syntax for declaring the varargs method uses ellipses (...) where the square brackets of an array might go.



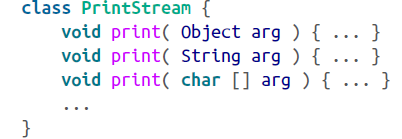




the compiler would automatically box the primitives to their wrapper types for us before placing them into the Object [].

**<9>Method Overloading(ad-hoc polymorphism)**

Method overloading is the ability to define multiple methods with the same name in a class; when the method is invoked, the compiler picks the correct one based on the arguments passed to the method. This implies that overloaded methods must have different numbers or types of arguments.



9: Object, String, char[], char, int, long, float, double, and boolean

What if we try to call print() with some other reference type? Say, a Date object? When there’s not an exact type match, the compiler searches for an acceptable, assignable match. Since Date, like all classes, is a subclass of Object, a Date object can be assigned to a variable of type Object.

What if there’s more than one possible match? For example, we try to print a subclass of String called MyString. (The String class is final so it can’t really be subclassed, but let’s use our imaginations.) MyString is assignable to either String or to Object. Here, the compiler makes a determination as to which match is “better” and selects that method. In this case, it’s the String method.

Method overloading is not something that happens at runtime; this is an important distinction. It means that the selected method is chosen once, when the code is compiled. This is in contrast to overridden methods, which are located at runtime and can be found even if they didn’t exist when the calling class was compiled.

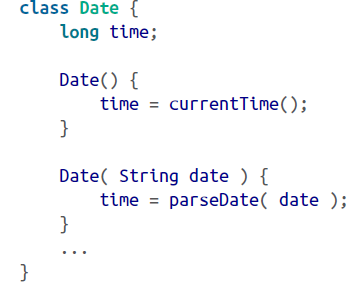
**3.Object Creation**

Objects in Java are allocated on a system “heap” memory space. Java takes care of memory allocation and deallocation for you. Java explicitly allocates storage for an object when you create it with the new operator. More importantly, objects are removed by garbage collection when they’re no longer referenced.

**<1>Constructors**

Objects are allocated with the new operator using an object constructor. A constructor is a special method with the same name as its class and no return type. Constructors, like other methods, can accept arguments and can be overloaded.

A few more notes: constructors can’t be declared abstract, synchronized, or final. Constructors can, however, be declared with the visibility modifiers public, private, or protected, just like other methods, to control their accessibility.



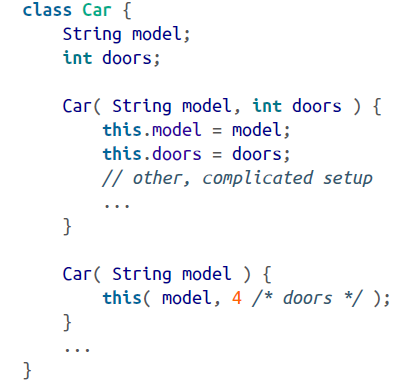
clipboard.png

clipboard.png

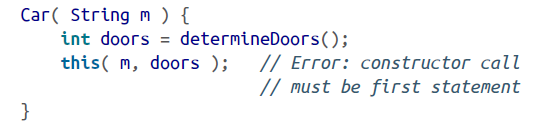
Setting this reference to null means it’s no longer pointing to the "Dec 25, 2006" string object. Setting the variable christmas to any other value would have the same effect. Unless the original string object is referenced by another variable, it’s now inaccessible and can be garbage-collected.

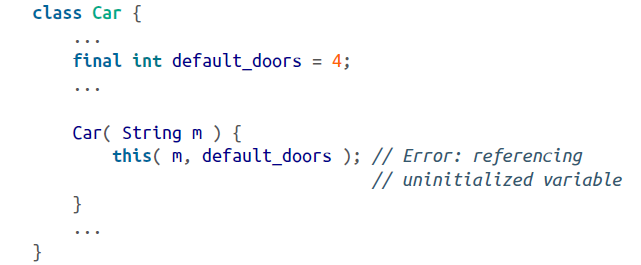
**<2>Working with Overloaded Constructors**

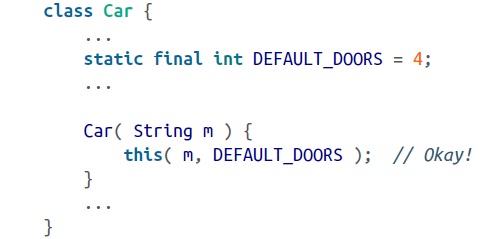
A constructor can invoke another overloaded constructor in its class using the self-referential method call this() with appropriate arguments to select the desired constructor.



The special call to this() must appear as the first statement in our delegating constructor. The syntax is restricted in this way because there’s a need to identify a clear chain of command in the calling of constructors. At the end of the chain, Java invokes the constructor of the superclass (if we don’t do it explicitly) to ensure that inherited members are initialized properly before we proceed.







You can invoke a second constructor (delegate to it) only as the first statement of your constructor.

**<3>Static and Nonstatic Initializer Blocks**

it’s executed once, at the time the object is constructed, or, in the case of a code block marked static, at the time the class is loaded. These blocks can be used to do additional setup for the class or an object instance and are called initializer blocks.

They’re called at the time the instance variable’s initializers are evaluated (after superclass construction, but before your constructor body), **in the order in which they appear in the Java source**:



Normally, this kind of setup could be done just as well in the object’s constructor. A notable exception is in the case of an anonymous inner class

Similarly, you can use static initializer blocks to set up static class members. This more useful case allows the static members of a class to have complex initialization just like objects do with constructors:



**4.Object Destruction**

**<1>Garbage Collection**

When all references to an object are gone and it’s no longer accessible, the garbage-collection mechanism declares the object unreachable and reclaims its space back to the available pool of resources. Modern Java garbage collectors effectively run continuously without forcing any lengthy delay in execution of the Java application. Because they are part of a runtime system, they can also accomplish some things that could not be done statically.

**Garbage collection algorithms**:

**mark and sweep**: Java first walks through the tree of all accessible object references and marks them as alive. Java then scans the heap, looking for identifiable objects that aren’t marked. In this technique, Java is able to find objects on the heap because they are stored in a characteristic way and have a particular signature of bits in their handles unlikely to be reproduced naturally. This kind of algorithm doesn’t become confused by the problem of cyclic references, in which objects can mutually reference each other and appear alive even when they are dead (Java handles this problem automatically). This scheme wasn’t the fastest method, however, and caused pauses in the program.

Sun’s Java implementation divides the memory heap into several areas for objects with different estimated lifespans. Short-lived objects are placed on a special part of the heap, which reduces the time to recycle them drastically. Objects that live longer can be moved to other, less volatile parts of the heap. In recent implementations, the garbage collector can even “tune” itself by adjusting the size of parts of the heap based on the actual application performance.

You can prompt the garbage collector to make a clean sweep explicitly by invoking the System.gc() method. This method is completely implementation-dependent and may do nothing, but it can be used if you want some guarantee that Java has cleaned up before you do an activity.

**<2>Finalization**

Before an object is removed by garbage collection, its finalize() method is invoked to give it a last opportunity to clean up its act and free other kinds of resources it may be holding. While the garbage collector can reclaim memory resources, it may not take care of things such as closing files and terminating network connections as gracefully or efficiently as could your code. That’s what the finalize() method is for. An object’s finalize() method is called once and only once before the object is garbage-collected. However, there’s no guarantee when that will happen. Garbage collection may, in theory, never run on a system that is not short of memory. It is also interesting to note that finalization and collection occur in two distinct phases of the garbage-collection process. First, items are finalized; then they are collected. It is, therefore, possible that finalization can (intentionally or unintentionally) create a lingering reference to the object in question, postponing its garbage collection. The object is, of course, subject to collection later if the reference goes away, but its finalize() method isn’t called again.

**<3>Weak and Soft References**

Java’s garbage collector reclaims objects when they are unreachable. An unreachable object, again, is one that is no longer referenced by any variables within your application and that is not reachable through any chain of references by any running thread. In some situations, however, it is advantageous to have Java’s garbage collector work with your application to decide when it is time to remove a particular object.

For these cases, Java allows you to hold an object reference indirectly through a special wrapper object, a type of java.lang.ref.Reference. If Java then decides to remove the object, the reference the wrapper holds turns to null automatically. While the reference exists,you may continue to use it in the ordinary way and, if you wish, assign it elsewhere (using normal references), preventing its garbage collection.

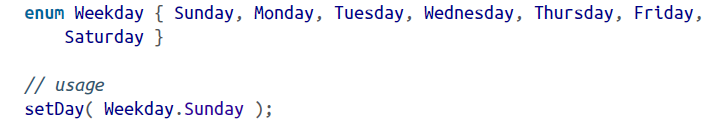
**two types of Reference wrappers(java.lang.ref):**

**WeakReference:** Weak references are eligible for garbage collection immediately; they do not prevent garbage collection the way that ordinary “strong” references do. This is an essential feature that allows garbage collection to work with certain kinds of caching schemes. You’ll often want to cache an object reference for performance (to avoid creating it or looking it up). But unless you take specific action to remove unneeded objects from your cache, the cache keeps those objects alive forever by maintaining live references to them. By using weak references, you can implement a cache that automatically throws away references when the object would normally be garbage-collected. In fact, an implementation of HashMap called WeakHashMap is provided that does just this.

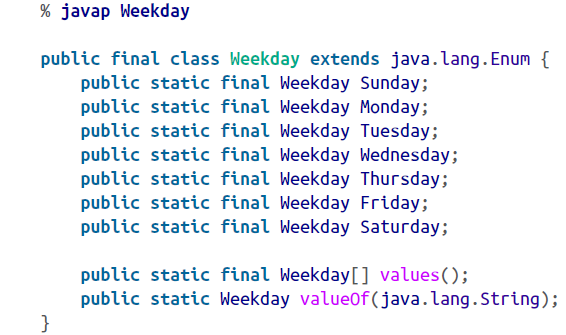
**SoftReference:** A soft reference is similar to a weak reference, but it tells the garbage collector to be less aggressive about reclaiming its contents. Soft-referenced objects are collected only when and if Java runs short of memory. This is useful for a slightly different kind of caching where you want to keep some content around unless there is a need to get rid of it.

**5.Enumerations**

An enumeration is an object type in the Java language that is limited to an explicit set of values. The values have an order that is defined by their order of declaration in the code, and have a correspondence with a string name that is the same as their declared name in the source code.(can also be used in switch statements)

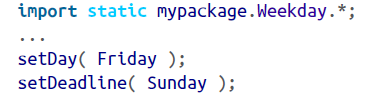


Let’s take a look at what the Java compiler is actually generating for the enum:



Weekday is a subclass of the Enum type with seven static, final, “constant” object references corresponding to our seven enumerated values. Each of the enumerated values is of type Weekday. The Java compiler does not let us extend this class or create any other instances of this type. The only instances of Weekday that will ever exist are the seven enumerated values. This is what gives enumerations their type safety.

Because enumerations are static values, they can be imported with the Java static import, saving us some typing:



We should also mention that enumerations can be declared not only at the “top level” alongside classes, but within classes or interfaces as well. In this case, they act just like inner classes.

**<1>Enum Values**

You can get the ordered list of enum values for a type with the static values() method.

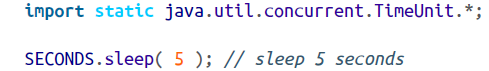
clipboard.png

We mentioned that enum values have a string correspondence for their names. You can get the string name of the value (which is exactly the same as it is declared in the source code) with the name() method. Going the other direction, you can “look up” any enum value by its class type and string name using the static Enum.valueOf() method:

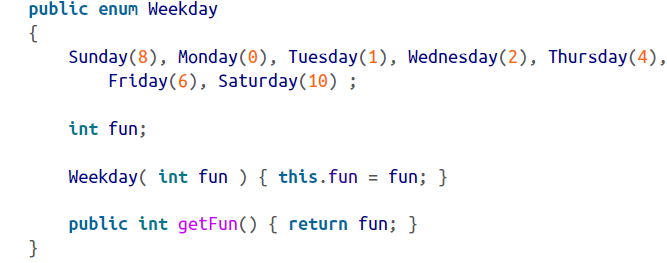
clipboard.png

**<2>Customizing Enumerations**

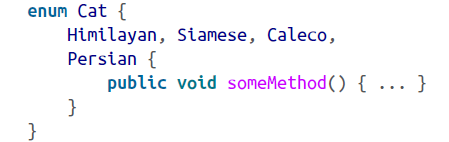
We said that the java.lang.Enum type cannot be directly extended and that you can’t create new instances of enum types. However, you can add things to the generated enumeration class when it’s declared.



Enumerations can have values with constructors, methods, and fields just like other classes.



Here, we’ve added an instance variable, fun, to the Weekday class, as well as a constructor and accessor method that work with the value. The declaration of our enum values each now accepts the constructor value, much like a constructor call without the new keyword. Note that the semicolon at the end of the values is mandatory. Each Weekday now has a fun attribute.



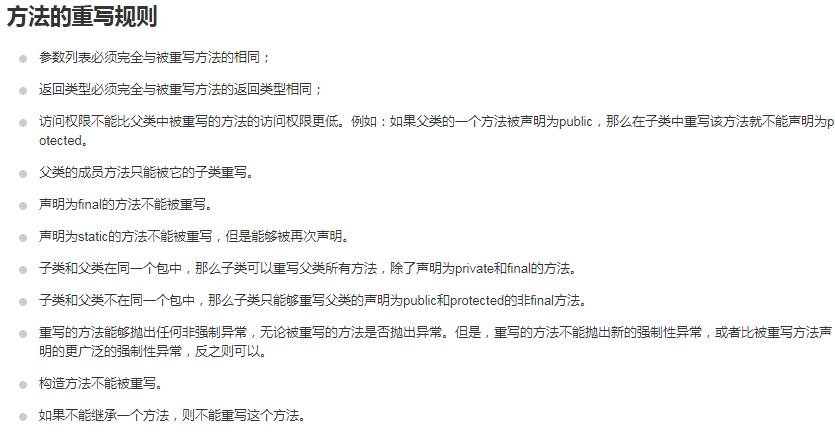
Now, only the Cat.Persian enum value has the method. In this case, the compiler generates a subclass of Cat as an inner class of the Persian type to hold the extra member.

**六. Relationships Among Classes**

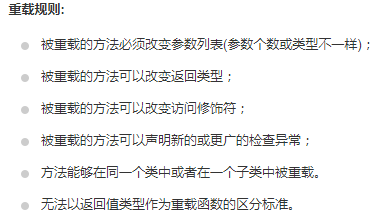
**Override(覆盖，重写) and Overload(重载):**

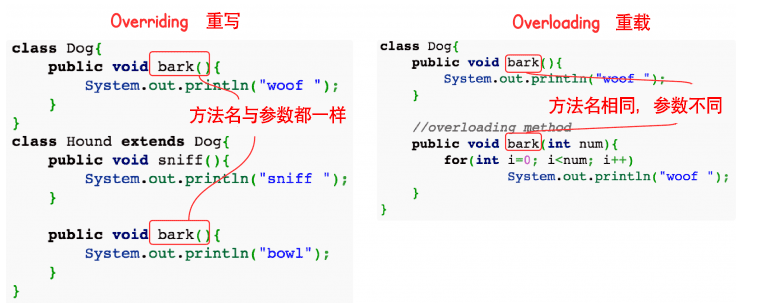
**Overloaded methods are selected by the compiler at compile time. Overridden methods, on the other hand, are selected dynamically at runtime.**

**Override:** 重写是子类对父类的允许访问的方法的实现过程进行重新编写, 返回值和形参都不能改变。**即外壳不变，核心重写！** 重写的好处在于子类可以根据需要，定义特定于自己的行为。 也就是说子类能够根据需要实现父类的方法。重写方法不能抛出新的检查异常或者比被重写方法申明更加宽泛的异常。例如： 父类的一个方法申明了一个检查异常 IOException，但是在重写这个方法的时候不能抛出 Exception 异常，因为 Exception 是 IOException 的父类，只能抛出 IOException 的子类异常。



**Overload:** 重载(overloading) 是在一个类里面，方法名字相同，而参数不同。返回类型可以相同也可以不同。每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。最常用的地方就是构造器的重载。

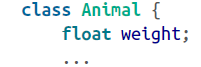


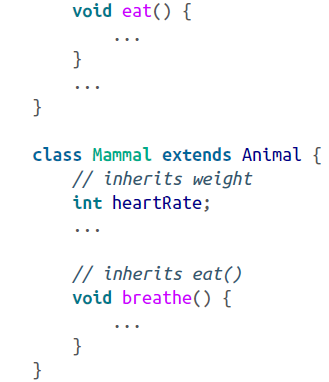


**1.Subclassing and Inheritance**

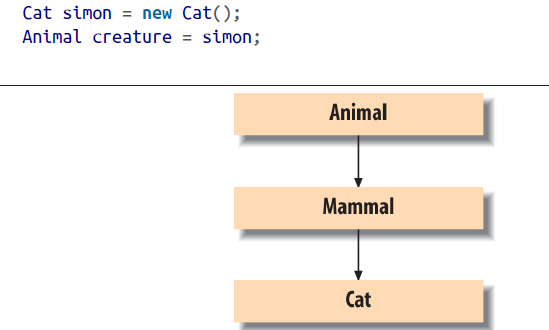
Classes in Java exist in a hierarchy. A class in Java can be declared as a subclass of another class using the extends keyword.

**A subclass inherits all members of its superclass not designated as private.** other levels of visibility affect which inherited members of the class can be seen from outside of the class and its subclasses, but at a minimum, a subclass always has the same set of visible members as its parent.



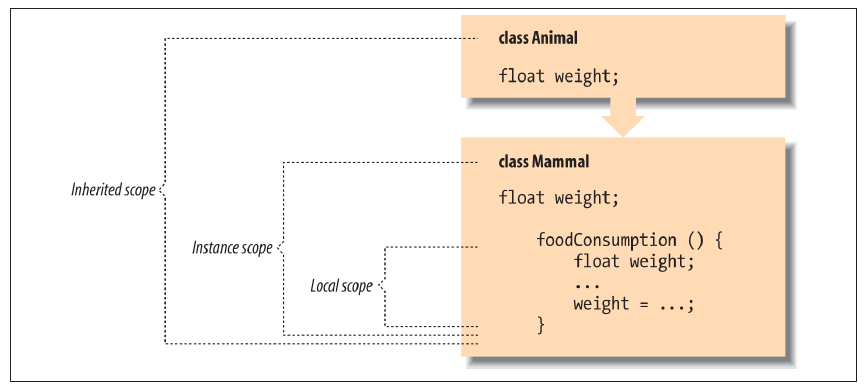


A class can extend only one other class. To use the proper terminology, Java allows single inheritance of class implementation. A subclass can be further subclassed. Normally, subclassing specializes or refines a class by adding variables and methods (you cannot remove or hide variables or methods by subclassing).



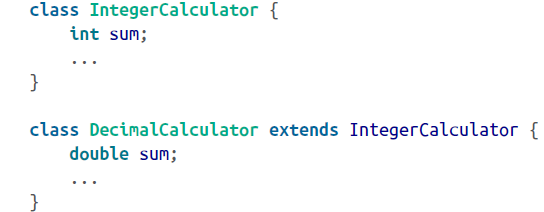
The Cat instance simon in this example can be assigned to the Animal type variable creature because Cat is a subtype of Animal. Similarly, any method accepting an Ani mal object would accept an instance of a Cat or any Mammal type as well. This is an important aspect of polymorphism in an object-oriented language such as Java.

**<1>Shadowed Variables**



A local variable of the same name as an instance variable shadows (hides) the instance variable. Similarly, an instance variable in a subclass can shadow an instance variable of the same name in its parent class.

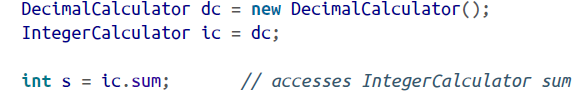
A slightly more plausible use of shadowed variables would involve changing their types. We can do this without changing the existing code because, as its name suggests, when we shadow variables, we don’t replace them but instead mask them. Both variables still exist; methods of the superclass see the original variable, and methods of the subclass see the new version. The determination of what variables the various methods see occurs at compile time.



In this example, we shadow the instance variable sum to change its type from int to double. **Methods defined in the class IntegerCalculator see the integer variable sum, while methods defined in DecimalCalculator see the floating-point variable sum. However, both variables actually exist for a given instance of DecimalCalculator, and they can have independent values. In fact, any methods that DecimalCalculator inherits from IntegerCalculator actually see the integer variable sum. Because both variables exist in DecimalCalculator, we need a way to reference the variable inherited from IntegerCalculator. We do that using the super keyword as a qualifier on the reference:**

clipboard.png

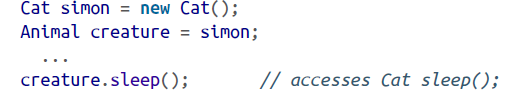
Another important point about shadowed variables has to do with how they work when we refer to an object by way of a less derived type (a parent type).For example, we can refer to a DecimalCalculator object as an IntegerCalculator by using it via a variable of type IntegerCalculator. If we do so and then access the variable sum, we get the integer variable, not the decimal one:



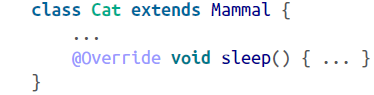
**<2>Overriding Methods**

We could declare overloaded methods (i.e., methods with the same name but a different number or type of arguments) within a class. A subclass can define additional overloaded methods that add to the overloaded methods provided by a superclass. A subclass can define a method that has exactly the same method signature (name and argument types) as a method in its superclass. In that case, the method in the subclass overrides the method in the superclass and effectively replaces its implementation.

From what you’ve seen so far, overridden methods probably look like they shadow methods in superclasses, just as variables do. But overridden methods are actually more powerful than that. When there are multiple implementations of a method in the inheritance hierarchy of an object, the one in the “most derived” class (the furthest down the hierarchy) always overrides the others, even if we refer to the object through a reference of one of the superclass types.



@Override : A common programming error in Java is to accidentally overload a method when trying to override it. Any difference in the number or type of arguments (the method signature) produces two overloaded methods instead of a single, overridden method. @Override tells the compiler that the method it marks is intended to override a method in the superclass. The compiler then warns if the method doesn’t match.

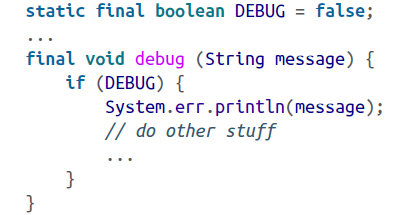


**Overriden methods and dynamic binding**: **Overloaded methods are selected by the compiler at compile time. Overridden methods, on the other hand, are selected dynamically at runtime.** If we created a new class that implements an additional, more specific,overloaded method, and replace the compiled class in our classpath with it, our code would continue to use the implementation it discovered originally. This situation would persist until we recompiled our code along with the new class.

**Static method binding**: Static methods don’t belong to any object instance; they are accessed directly through a class name. they are always bound at compile time. A static method in a superclass can be shadowed by another static method in a subclass, as long as the original method was not declared final. However, both methods are always accessible directly via their respective class names.

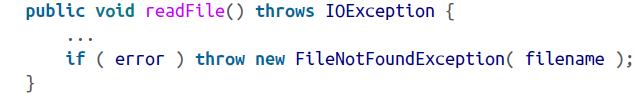
**final methods and performance**: In Java, instance methods are, by default, dynamic. But you can use the final modifier to declare that an instance method can’t be overridden in a subclass, and it won’t be subject to dynamic binding. We have seen final used with variables to effectively make them constants. When applied to a method, final means that its implementation is constant—no overriding allowed. final can also be applied to an entire class, which means the class can’t be subclassed. A profiling runtime can determine which methods are not being overridden and “optimistically” inline them, treating them as if they were final until it becomes necessary to do otherwise. As a rule, you should use the final keyword when it is correct for your program’s structure, not for performance considerations.

**Compiler optimizations**: In some older versions of Java, the javac compiler can be run with a -O switch, which tells it to perform certain optimizations, like inlining, statically. Most of these optimizations are now done at runtime by smarter VMs. The Java compiler is smart enough to remove this code when it determines that it won’t be called. For example:

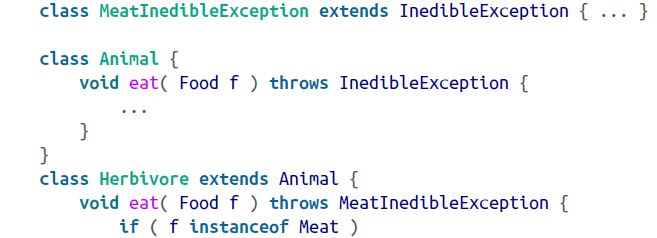


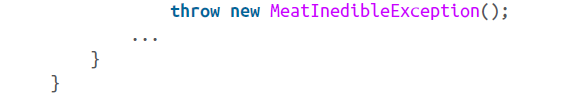
**Method selection revisited**:A method is considered more specific than another if its arguments are assignable to the arguments of the second method. We can now expand this rule to include the resolution of overridden methods by adding the following condition: to be more specific than another method, the type of the class containing the method must also be assignable to the type of the class holding the second method.

**Exceptions and overriden methods**: An overriding method may change the behavior of an object, but in some ways, it must still fulfill the contract of the original method with the user. Specifically, an overriding method must adhere to the throws clause of the original method. The new method cannot throw new types of checked exceptions. It can only declare that it throws exception types assignable to those thrown by the method in the parent class; that is, it may declare that it throws the same types of exceptions or subtypes of those declared by the original method. If the new method does not throw any of the checked exceptions of the original, it does not have to declare them and callers of the method via the subclass do not have to guard against them.



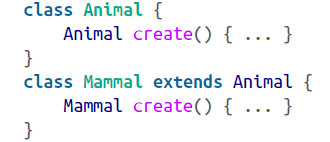
When we override a method in a subclass, we get an opportunity to rewrite the throws clause of the method a bit. The new method must still be backward-compatible with the original, so any checked exceptions it throws must be assignable to those thrown by the overridden method.





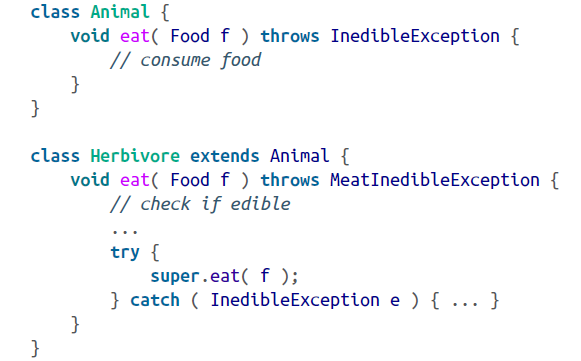
To sum up, an overriding method can refine not only the behavior of the parent method, but also the type of checked exceptions it throws.

**Return types and overridden methods**: when you override a method, you may change the return type to a subtype of the original method’s return type.



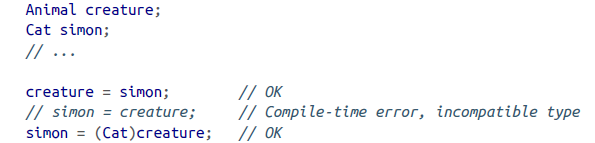
**<3>Special Refernces: this and super**

The special references this and super allow you to refer to the members of the current object instance or to members of the superclass, respectively.

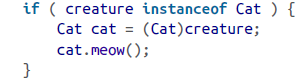


**<4>Casting**

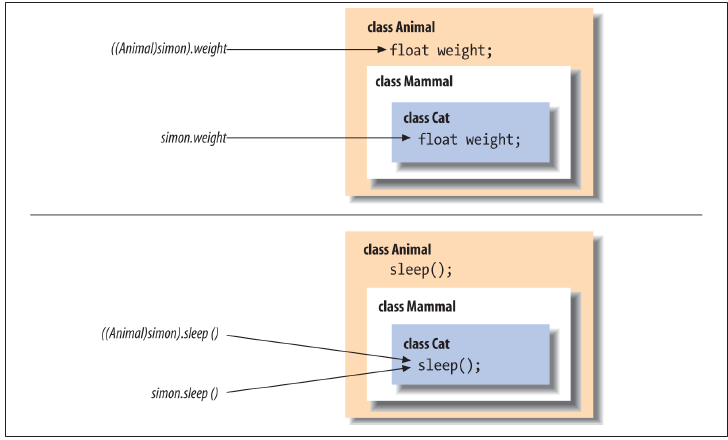
A cast explicitly tells the compiler to change the apparent type of an object reference. The main use for casts is when an object is temporarily assigned to a more general type. In Java, casts are checked both at compile time and at runtime to make sure they are legal. Attempting to cast an object to an incompatible type at runtime results in a ClassCastException. Casts in Java affect only the treatment of references; they never change the form of the actual object.



A cast can be used to narrow or downcast the type of a reference—to make it more specific. Often, we’ll do this when we have to retrieve an object from a more general type of collection or when it has been previously used as a less derived type. If you aren’t sure whether the Object is a Cat or a Dog at runtime, you can check it with instanceof before you perform the cast. If you do not check and you get the cast wrong, the runtime system throws a ClassCastException.



Casting can affect the selection of compile-time items such as variables and overloaded methods, but not the selection of overridden methods. As shown in the top half of the diagram, casting the reference simon to type Animal (widening it) affects the selection of the shadowed variable weight within it. However, as the lower half of the diagram indicates, the cast doesn’t affect the selection of the overridden method sleep().

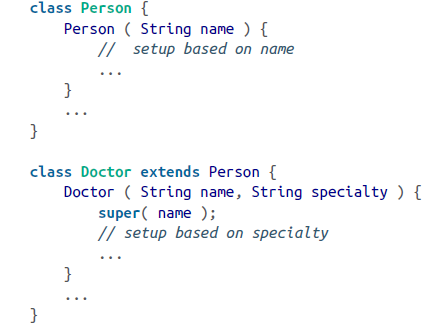


Generics

**<5>Using Superclass Constructors**

super() explicitly invokes the constructor of a superclass.

If we are going to call a superclass constructor with super(), it must be the first statement of our constructor, just as this() must be the first call we make in an overloaded constructor.



if you subclass a class whose constructors all take arguments, you have to invoke one of the superclass’s constructors explicitly from at least one of your subclass’s constructors.

**<6>Full Disclosure: Constructors and Initialization**

We can now tell the full story of how constructors are chained together and when instance variable initialization occurs:

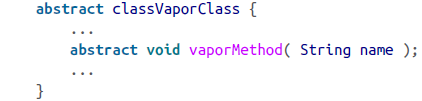
a> If the first statement of a constructor is an ordinary statement—that is, not a call to this() or super()—Java inserts an implicit call to super() to invoke the default constructor of the superclass. Upon returning from that call, Java initializes the instance variables of the current class and proceeds to execute the statements of the current constructor.

b> If the first statement of a constructor is a call to a superclass constructor via super(), Java invokes the selected superclass constructor. Upon its return, Java initializes the current class’s instance variables and proceeds with the statements of the current constructor.

c> If the first statement of a constructor is a call to an overloaded constructor via this(), Java invokes the selected constructor, and upon its return, simply proceeds with the statements of the current constructor. The call to the superclass’s constructor has happened within the overloaded constructor, either explicitly or implicitly, so the initialization of instance variables has already occurred.

**<7>Abstract Methods and Classes**

A method in Java can be declared with the abstract modifier to indicate that it’s just a prototype. An abstract method has no body. You can’t directly use a class that contains an abstract method; you must instead create a subclass that implements the abstract method’s body. In Java, a class that contains one or more abstract methods must be explicitly declared as an abstract class.

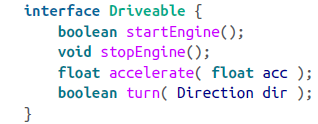


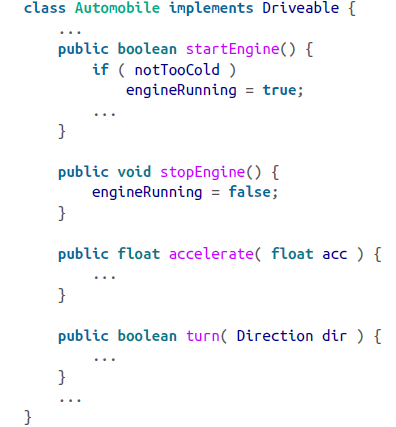
An abstract class can contain other nonabstract methods and ordinary variable declarations, but it can’t be instantiated. To be used, it must be subclassed and its abstract methods must be “overridden” with methods that implement a body. Not all abstract methods have to be implemented in a single subclass, but a subclass that doesn’t override all its superclass’s abstract methods with actual, concrete implementations must also be declared abstract.

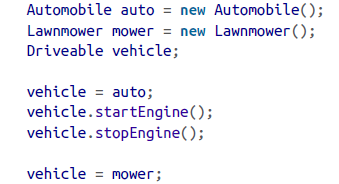
**2.Interfaces**

Java expands on the concept of abstract methods with interfaces. Unlike extending an abstract class, a class implementing an interface doesn’t have to inherit from any particular part of the inheritance hierarchy or use a particular implementation. Interface types act like class types. A class can implement as many interfaces as it desires. In this way, interfaces in Java replace much of the need for multiple inheritance in other languages.

You can declare variables to be of an interface type, you can declare arguments of methods to accept interface types, and you can specify that the return type of a method is an interface type. In each case, what is meant is that any object that implements the interface (i.e., wears the right merit badge) can fill that role.





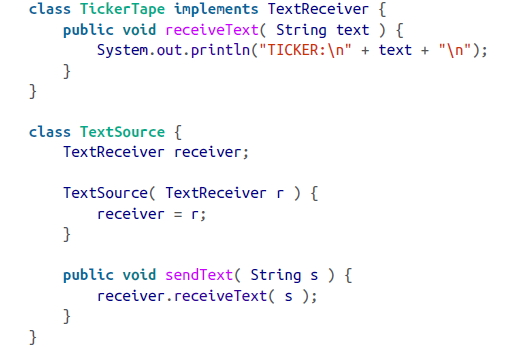


clipboard.png

**<1>Interfaces as Callbacks**

This is when an object effectively passes a reference to one or more of its methods to another object. The callback occurs when the called object subsequently invokes one of the methods. More generally, this concept is extended in Java to the concept of events in which listener objects register with event sources.

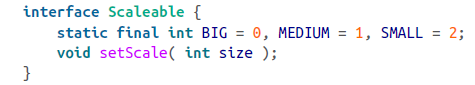
clipboard.png

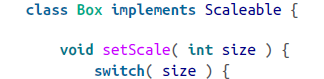


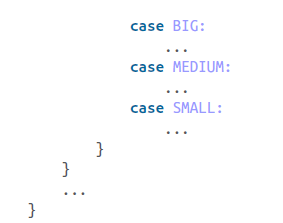
a TickerTape class that displays data and a TextSource class that provides an information feed. We’d like our TextSource to send any new text data. We could have TextSource store a reference to a TickerTape object, but then we could never use our TextSource to send data to any other kind of object. Instead, we’d have to proliferate subclasses of TextSource that dealt with different types. A more elegant solution is to have TextSource store a reference to an interface type, TextReceiver.

**<2>Interface Variables**

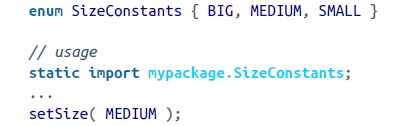
Although interfaces mostly allow us to specify behavior without implementation, there’s one exception. An interface can contain constants (static final variables ), which can be referred to directly through the interface name, and which also appear in any class that implements the interface.







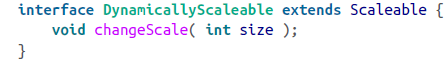
All variables defined in interfaces are implicitly final and static; you don’t need to use the modifiers, but for clarity, we recommend that you do. A class that implements Scaleable sees these constants. The main incentive for doing so disappeared when Java added enumerations and static imports.



**Flag interfaces:** Sometimes completely empty interfaces serve as a marker that a class has a special property. The java.io.Serializeable interface is a good example. Classes that implement Serializeable don’t have to add any methods or variables. Their additional type simply identifies them to Java as classes that want to be able to be serialized.

**<3>Subinterfaces**

An interface can extend another interface, just as a class can extend another class.



The interface DynamicallyScaleable extends our previous Scaleable interface and adds an additional method. A class that implements DynamicallyScaleable must implement all the methods of both interfaces. If you want to extend two or more interfaces, list them after the extends keyword, separated by commas:

clipboard.png

**Overlapping and conflicting methods**: If two methods in different interfaces have exactly the same signature and return type, there is no problem and the implementation in the class satisfies both interfaces. If the methods differ in the way that overloaded methods do, the class must implement both method signatures. If the methods have the same name but differ in return or exception types, the class cannot implement both and compile-time errors occur.

**3.Packages and Compilation Units**

A package is a name for a group of related classes and interfaces. In this sense, packages are somewhat like libraries; they organize and manage sets of classes.

**<1>Compilation Units**

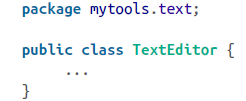
The division of classes into their own files is important because the Java compiler assumes much of the responsibility of a make or build utility. The compiler relies on the names of source files to find and compile dependent classes. It’s possible to put more than one class definition into a single file, but there are some restrictions.

**<2>Package Names**

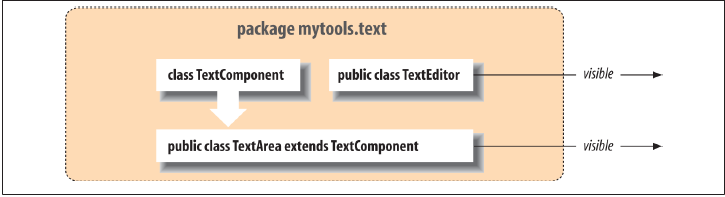
Package names are hierarchical in nature, using a dot-separated naming convention. By default, package name components correspond to directory names and serve as a unique path for the compiler and runtime systems to locate Java source files and classes. if we create another package called mytools.text.poetry, those classes won’t be part of the mytools.text package; they won’t have the access privileges of package members.

**<3>Class Visibility**

By default, a class is accessible only to other classes within its package. To be used outside of its package, a class must be declared as public.

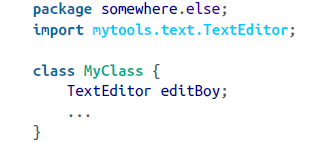


The class TextEditor can now be referenced anywhere. A Java source code file can have only a single public class defined within it and the file must be named for that class.



The classes TextArea and TextEditor are declared public so that they can be used elsewhere in an application. The class TextComponent is part of the implementation of TextArea and is not accessible from outside of the package.

**<4>Importing Classes**



It is also possible to import all the classes in a package using the \* wildcard notation:

clipboard.png

There can be a problem with importing classes that have conflicting names. The compiler prevents you from explicitly importing two classes with the same name and gives you an error if you try to use an ambiguous class that could come from two packages imported with the package import notation. Other than the potential for naming conflicts, there’s no penalty for importing many

classes. Java doesn’t carry extra baggage into the compiled class files. In other words, Java class files don’t contain information about the imports; they only reference classes actually used in them.

If you are going to use more than two or three classes from a package, consider the package import.

The unnamed package:

**4.Visibility of Variables and Methods**

One of the most important aspects of object-oriented design is *data hiding*, or *encapsulation*.

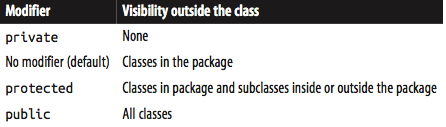
**(1)Basic Access Modifiers**

**friendly(default): variables and methods are accessible to members of the class iteself and other classes in the same pkg;**

**private: only accessible by members within the class;**

**public: any class in any pkg;**

protected: slightly less restrictive than the default level(friendly). Protected members are visible to subclasses of the class, even if they are defined in a different package.(different to C++)



**(2)Subclasses and Visibility**

**in Java:**

**overide/overwrite: 子类覆盖父类**

**overload: 重载，几个函数用同一个名字表示，参数不同**

**in C++:**

**override(覆盖): 子类覆盖父类，父类函数必须有virtual关键字**

**overwrite(重写): 子类重写父类函数，父类的函数不管有没有virtual都被隐藏了**

**overload(重载): 和java一样**

**visibility: subclass can only increase the visibility of the method in super class, can't reduce it; (子类覆盖父类方法时只能扩大其可见度，不能缩小)。eg: 父类private ，子类可以public，反之不行；(**If we could override a method with a less visible method, we would have a problem: our Mammal might not be able to do all the things an Animal can.**)** However, we can reduce the visibility of a variable. In this case, the variable acts like any other shadowed variable; the two variables are distinct and can have separate visibilities in different classes.**(变量可以缩小可见度);**

**protected:** he protected variables of a class are visible to its subclasses, but only through objects of the subclass’s type or its subtypes.A subclass can see a protected variable of its superclass as an inherited variable, but it can’t access that same variable via a reference to the superclass itself. **(protected的变量，子类访问父类的protected只能通过继承的方式，不能直接引用)。** Visibility modifiers don’t restrict access between instances of the same class in the same way that they restrict access between instances of different classes. Two instances of the same class can access all of each other’s members, *including private ones*, as long as they refer to each other as the correct type. (visibility modifier 只能用来限定不同class的对象，同一class的不同实例不受限制)

**(3)Interfaces and Visibility**

Interfaces behave like classes within packages. An interface can be declared public to make it visible outside its package. Under the default visibility, an interface is visible only inside its package. Like classes, only one public interface can be declared in a compilation unit (file).

**5.Arrays and the Class Hierarchy**

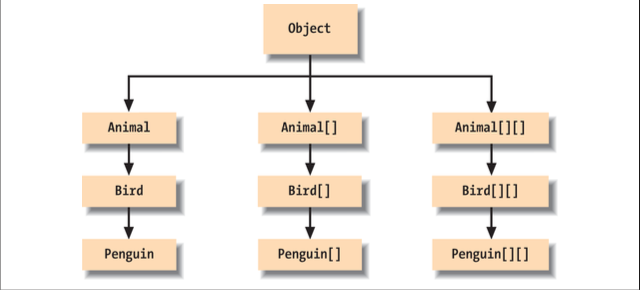


image.png

**6.Inner Classes**

**inner class/ anonymous inner class: play part of the role of closures, are pure syntactic sugar.**

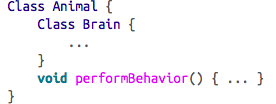


image.png

**inner class can be declared as static.**

**(1)Inner classes as Adapters**

**(2)Inner classes Within Methods**