The Java Language Specification

Release 1.0 Alpha2

Sun Microsystems Computer Corporation
A Sun Microsystems, Inc. Business

March 15, 1995

Contents

Java Language Specification 1

1 Program Structure 1

2 Lexical Issues 1
- 2.1 Comments 1
- 2.2 Identifiers 2
- 2.3 Keywords 2
- 2.4 Literals 3
- 2.5 Operators and Miscellaneous Separators 4

3 Types 4
- 3.1 Numeric Types 5
- 3.2 Boolean Types 6
- 3.3 Arrays 6

4 Classes 7
- 4.1 Casting Between Class Types 8
- 4.2 Methods 9
- 4.3 Overriding Methods 11
- 4.4 Overload Resolution 11
- 4.5 Constructors 12
- 4.6 Object Creation—the **new** Operator 13
- 4.7 Static Methods, Variables, and Initializers 14
- 4.8 Access Specifiers 16
- 4.9 Variable Scoping Rules 17
- 4.10 Modifiers 17

5 Interfaces 19
- 5.1 Interfaces as Types 20
- 5.2 Methods in Interfaces 20
- 5.3 Variables in Interfaces 21
- 5.4 Combining Interfaces 21

6 Packages 21
- 6.1 Specifying a Compilation Unit's Package 21
- 6.2 Using Classes and Interfaces from Other Packages 21

7 Expressions 22
- 7.1 Operators 22
- 7.2 Casts and Conversions 25

8 Statements 25
8.1 Declarations 25
8.2 Expressions 26
8.3 Control Flow 26
8.4 Exceptions 27

A Appendix: Floating Point 33
A.1 Special Values 33
A.2 Binary Format Conversion 33
A.3 Ordering 34
A.4 Summary of IEEE-754 Differences 34

Java Language Specification

This document is a preliminary specification of the Java™ language. Both the specification and the language are subject to change. When a feature that exists in both Java and ANSI C isn't explained fully in this specification, the feature should be assumed to work as it does in ANSI C. Send comments on the Java Language and specification to WebRunner@Sun.COM.

1 *Program Structure*

The source code for an Java program consists of one or more *compilation units*. Each compilation unit can contain only the following (in addition to white space and comments):

- a package statement (see "Packages" on page 21)
- import statements (see "Packages" on page 21)
- class declarations (see "Classes" on page 7)
- interface declarations (see "Interfaces" on page 19)

Although each Java compilation unit can contain multiple classes or interfaces, at most one class or interface per compilation unit can be public (see "Classes" on page 7).

When Java source code is compiled, the result is Java bytecode. Java bytecode consists of machine-independent instructions that can be interpreted efficiently by the Java runtime system. The Java runtime system operates like a virtual machine, for information see *The Java Virtual Machine Specification*.

Implementation Note: In the current Java implementation, each compilation unit is a file with a ".java" suffix.

2 *Lexical Issues*

During compilation, the characters in Java source code are reduced to a series of tokens. The Java compiler recognizes five kinds of tokens: identifiers, keywords, literals, operators, and miscellaneous separators. Comments and *white space* such as blanks, tabs, line feeds, and are not tokens, but they often are used to separate tokens.

Java programs are written using the Unicode character set or some character set that is converted to Unicode before being compiled.

2.1 Comments

The Java language has three styles of comments:

`// text`	All characters from **//** to the end of the line are ignored.
`/* text */`	All characters from **/*** to ***/** are ignored.
`/** text */`	These comments are treated specially when they occur immediately before any declaration. They are illegal in any other place in the code. These comments indicate that the enclosed text should be included in automatically generated documentation as a description of the declared item.

2.2 Identifiers

Identifiers must start with a letter, underscore ("_"), or dollar sign ("`$`"); subsequent characters can also contain digits (`0-9`). Java uses the Unicode character set. For the purposes of determining what is a legal identifier the following are considered "letters:"

- The characters "`A`" through "`Z`"
- The characters "`a`" through "`z`"
- All Unicode characters with a character number above hex `00C0`

Other characters valid after the first letter of an identifier include every character except those in the segment of Unicode reserved for special characters.

Thus, "`garçon`" and "`Mjølner`" are legal identifiers, but strings containing characters such as "¶" are not.

For more information on the Unicode standard, see *The Unicode Standard, Worldwide Character Encoding*, Version 1.0, Volumes 1&2. The FTP address for Unicode, Inc. (formerly the Unicode Consortium) is unicode.org.

2.3 Keywords

The following identifiers are reserved for use as keywords. They cannot be used in any other way.

abstract	default	goto[a]	null	synchronized
boolean	do	if	package	this
break	double	implements	private	threadsafe
byte	else	import	protected	throw
byvalue[a]	extends	instanceof	public	transient
case	false	int	return	true
catch	final	interface	short	try
char	finally	long	static	void
class	float	native	super	while
const[a]	for	new	switch	
continue				

a. Reserved but currently unused.

2.4 Literals

Literals are the basic representation of any integer, floating point, boolean, character, or string value.

2.4.1 *Integer Literals*

Integers can be expressed in decimal (base 10), hexadecimal (base 16), or octal (base 8) format. A decimal integer literal consists of a sequence of digits (optionally suffixed as described below) *without* a leading 0 (zero). An integer can be expressed in octal or hexadecimal rather than decimal. A leading 0 (zero) on an integer literal means it is in octal; a leading 0x (or 0X) means hexadecimal. Hexadecimal integers can include digits (0-9) and the letters a-f and A-F. Octal integers can include only the digits 0-7.

Integer literals are of type **int** unless they are larger than 32-bits, in which case they are of type **long** (see “Integer Types” on page 5). A literal can be forced to be **long** by appending an `L` or `l` to its value.

The following are all legal integer literals:

```
2, 2L  0777  0xDeadBeef
```

2.4.2 *Floating Point Literals*

A floating point literal can have the following parts: a decimal integer, a decimal point (“.”), a fraction (another decimal number), an exponent, and a type suffix. The exponent part is an `e` or `E` followed by an integer, which can be signed. A floating point literal must have at least one digit, plus either a decimal point or `e` (or `E`). Some examples of floating point literals are:

```
3.1415 3.1E12  .1e12  2E12
```

As described in “Floating Point Types” on page 5, the Java language has two floating point types: **float** (IEEE 754 single precision) and **double** (IEEE 754 double precision). You specify the type of a floating point literal as follows:

```
2.0d or 2.0D         double
2.0f or 2.0F or 2.0  float
```

2.4.3 *Boolean Literals*

The **boolean** type has two literal values: `true` and `false`. See “Boolean Types” on page 6 for more information on boolean values.

2.4.4 *Character Literals*

A character literal is a character (or group of characters representing a single character) enclosed in single quotes. Characters have type **char** and are drawn from the Unicode character set (see “Character Types” on page 5). The following escape sequences allow for the representation of some non-graphic characters as well as the single quote, "'" and the backslash "\", in Java code:

continuation	<newline>	\
new-line	NL (LF)	\n
horizontal tab	HT	\t
back space	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
single quote	'	\'
double quote	"	\"
octal bit pattern	0ddd	\ddd
hex bit pattern	0xdd	\xdd
unicode char	0xdddd	\udddd

2.4.5 *String Literals*

A string literal is zero or more characters enclosed in double quotes. Each string literal is implemented as a String object (*not* as an array of characters). For example, "abc" creates an new instance of class String. The following are all legal string literals:

```
"" \\ the empty string
"\""
"This is a string"
"This is a \
   two-line string"
```

2.5 **Operators and Miscellaneous Separators**

The following characters are used in source code as operators or separators:

```
+  -  !  %  ^  &  *  |  ~  /  >  <

(  )  {  }  [  ]  ;  ?  :  ,  .  =
```

In addition, the following character combinations are used as operators:

```
++   --   ==   <=   >=   !=   <<   >>

>>>   +=   - =   *=   /=   &=   |=

^=   %=   <<=   >>=   >>>=   ||   &&
```

For more information see "Operators" on page 22.

3 *Types*

Every variable and every expression has a type. Type determines the allowable range of values a variable can hold, allowable operations on those values, and the meanings of the operations. Built-in types are provided by the Java language. Programmers can compose new types using the *class* and *interface* mechanisms (see “Classes” on page 7 and “Interfaces” on page 19).

The Java language has two kinds of types: simple and composite. Simple types are those that cannot be broken down; they are atomic. The integer, floating point, boolean, and character types are all simple types. Composite types are built on simple types. The language has three kinds of composite types: arrays, classes, and interfaces. Simple types and arrays are discussed in this section.

3.1 Numeric Types

3.1.1 *Integer Types*

Integers are similar to those in C and C++, with two exceptions: all integer types are machine independent, and some of the traditional definitions have been changed to reflect changes in the world since C was introduced. The four integer types have widths of 8, 16, 32, and 64 bits and are signed.

Width	Name
8	byte
16	short
32	int
64	long

A variable’s type does not directly affect its storage allocation. Type only determines a variable’s arithmetic properties and legal range of values. If a value is assigned to a variable that is outside the legal range of the variable, the value is reduced modulo the range.

3.1.2 *Floating Point Types*

The **float** keyword denotes single precision (32 bit); **double** denotes double precision (64 bit). The result of a binary operator on two **float** operands is a **float**. If either operand is a **double**, the result is a **double**.

Floating point arithmetic and data formats are defined by IEEE 754. See “Appendix: Floating Point” on page 33 for details on the floating point implementation.

3.1.3 *Character Types*

The language uses the Unicode character set throughout. Consequently the **char** data type is defined as a 16-bit unsigned integer.

3.2 Boolean Types

The **boolean** type is used for variables that can be either **true** or **false**, and for methods that return **true** and **false** values. It's also the type that is returned by the relational operators (e.g., ">=").

Boolean values are not numbers and cannot be converted into numbers by casting.

3.3 Arrays

Arrays in the language are first class objects. They replace pointer arithmetic. All objects (including arrays) are referred to by pointers that cannot be damaged by being manipulated as numbers. Arrays are created using the **new** operator:

```
char s[] = new char[30];
```

The first element of an array is at index 0 (zero). Specifying dimensions in the declarations is not allowed. Every allocation of an array must be explicit—use **new** every time:

```
int i[] = new int[3];
```

The language does not support multi-dimensional arrays. Instead, programmers can create arrays of arrays:

```
int i[][] = new int[3][4];
```

At least one dimension must be specified but other dimensions can be explicitly allocated by a program at a later time. For example:

```
int i[][] = new int[3][];
```

is a legal declaration.

Subscripts are checked to make sure they're valid:

```
int a[] = new int[10];
a[5] = 1;
a[1] = a[0] + a[2];
a[-1]   = 4;         // Throws an ArrayIndexOutOfBoundsException
                     // at runtime
a[10] = 2;           // Throws an ArrayIndexOutOfBoundsException
                     // at runtime
```

Array dimensions must be integer expressions:

```
int n;
...
float arr[] = new float[n + 1];
```

The length of any array can be found by using **.length**:

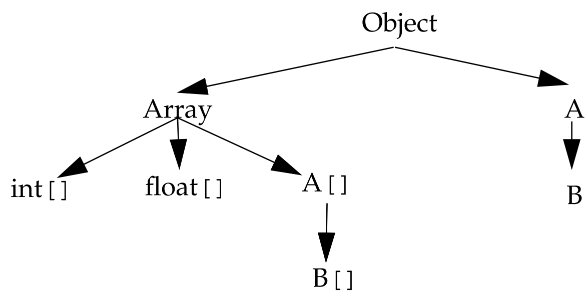
```
int a[][] = new int[10][3];
println(a.length);           // prints 10
println(a[0].length);        // prints 3
```

3.3.1 *Array Detail*

Arrays are instances of subclasses of class Object. In the class hierarchy there is a class named Array, which has one instance variable, "length". For each primitive type there is a corresponding subclass of Array. Similarly, for all classes a corresponding subclass of Array implicitly exists. For example:

```
new Thread[n]
```

creates an instance of Thread `[]`. If class A is a superclass of class B (i.e., B extends A) then A `[]` is a superclass of B `[]` (see the diagram below).



Hence, you can assign an array to an Object:

```
Object o;
int a[] = new int[10];
o = a;
```

and you can cast an Object to an array:

```
a = (int[])o;
```

Array classes cannot be explicitly subclassed.

4 *Classes*

Classes represent the classical object oriented programming model. They support data abstraction and implementations tied to data. In Java, each new class creates a new type.

To make a new class, the programmer must base it on an existing class. The new class is said to be *derived* from the existing class. The derived class is also called a *subclass* of the other, which is known as a *superclass*. Class derivation is transitive: if B is a subclass of A, and C is a subclass of B, then C is a subclass of A.

The immediate superclass of a class and the interfaces (see “Interfaces” on page 19) that the class implements (if any) are indicated in the class declaration by the keywords **extends** and **implements**, respectively:

[Doc comment] [Modifiers] **class** *Classname*
    extends *Superclassname]*
    implements *Interface{, Interface}] {*
        ClassBody
}

For example:

```
/** 2 dimensional point */
public class Point {
    float x, y;
    ...
}

/** Printable point */
class PrintablePoint extends Points implements Printable {
    ...
    public void print() {
        ...
    }
}
```

All classes are derived from a single root class: Object. Every class except Object has exactly one immediate superclass. If a class is declared without specifying an immediate superclass, Object is assumed. For example, the following:

```
class Point {
    float x, y;
}
```

is the same as:

```
class Point extends Object {
    float x, y;
}
```

The language supports only single inheritance. Through a feature known as *interfaces*, it supports some features that in other languages are supported through multiple inheritance (see “Interfaces” on page 19).

4.1 Casting Between Class Types

The language supports casting between types and because each class is a new type, Java supports casting between class types. If B is a subclass of A, then an instance of B can be used as an instance of A. No explicit cast is required, but an explicit cast is legal—this is called *widening*. If an instance of A needs to be used as if it were an instance of B, the programmer can write a type conversion or *cast*—this is called *narrowing*. Casts from a class to a subclass are always checked at runtime to make sure that the object is actually an instance of the subclass (or one of its subclasses). Casting between sibling classes is a compile-time error. The syntax of a class cast is:

```
(classname)ref
```

where `(classname)` is the object being cast to and `ref` is the object being cast.

Casting affects only the reference to the object, not the object itself. However, access to instance variables is affected by the type of the object reference. Casting an object from one type to another may result in a different instance variable being accessed even though the same variable name is used.

```
class ClassA {
    String name = "ClassA";
}

class ClassB extends ClassA {  // ClassB is a subclass of ClassA
    String name= “ClassB”;
}
```

```
class AccessTest {
    void test() {
        ClassB b = new ClassB();
        println(b.name);         // print: ClassB

        ClassA a;
        a = (ClassA)b;
        println(a.name);         // print: ClassA
    }
}
```

4.2 Methods

Methods are the operations that can be performed on an object or class. They can be declared in either classes or interfaces, but they can be implemented only in classes. (All user-defined operations in the language are implemented with methods.)

A method declaration in a class has the following form (native and abstract methods have no method body):

[Doc comment] [Modifiers] returnType methodName (parameterList) {
    [methodBody]
}

Methods:

- Have a return type unless they're constructors, in which case they have no return type. If a non-constructor method does not return any value, it must have a **void** return type.
- Have a parameter list consisting of comma-separated pairs of types and parameter names. The parameter list should be empty if the method has no parameters.

Variables declared in methods (*local variables*) can't hide other local variables or parameters in the same method. For example, if a method is implemented with a parameter named `i`, it's a compile-time error for the method to declare a local variable named `i`. In the following example:

```
class Rectangle {
    void vertex(int i, int j) {
        for (int i = 0; i <= 100; i++) {// ERROR
        ...
        }
    }
}
```

the declaration of "`i`" in the for loop of the method body of "`vertex`" is a compile-time error.

The language allows *polymorphic* method naming—declaring a method with a name that has already been used in the class or its superclass—for overriding and overloading methods. *Overriding* means providing a different implementation of an inherited method. *Overloading* means declaring a method that has the same name as another method, but a different parameter list.

Note: Return types are not used to distinguish methods. Within a class scope, methods that have the same name and parameter list, i.e., the same number, position, and types of parameters, *must* return the same type. It is a compile-time error to declare such a method with a different return type.

4.2.1 *Instance Variables*

All variables in a class declared outside the scope of a method and not marked static (see “Static Methods, Variables, and Initializers” on page 14) are instance variables. (Variables declared inside the scope of a method are considered local variables.) Instance variables can have modifiers (see “Modifiers” on page 17).

Instance variables can be of any type and can have initializers. (If an instance variable does not have an initializer, it is initialized to zero; **boolean** variables are initialized to **false**; and objects are initialized to **null**. An example of an initializer for an instance variable named **j** is:

```
class A {
    int j = 23;
    ...
}
```

4.2.2 *The this and super Variables*

Inside the scope of a non-static method, the name **this** represents the current object. For example, an object may need to pass itself as an argument to another object’s method:

```
class MyClass {
    void aMethod(OtherClass obj) {
        ...
        obj.Method(this);
            ...
    }
}
```

Any time a method refers to its own instance variables or methods, an implicit “`this.`” is in front of each reference:

```
class Foo {
    int a, b, c;
        ...
    void myPrint(){
        print(a + "\n");      // a == "this.a"
    }
        ...
}
```

The **super** variable is similar to the **this** variable. The **this** variable contains a reference to the current object; its type is the class containing the currently executing method. The **super** variable contains a reference which has the type of the superclass.

4.2.3 *Setting Local Variables*

Methods are rigorously checked to be sure that all *local variables* (variables declared inside a method) are set before they are referenced. Using a local variable before it is initialized is a compile-time error.

4.3 Overriding Methods

To override a method, a subclass of the class that originally declared the method must declare a method with the same name, return type (or a subclass), and parameter list. When the method is invoked on an instance of the subclass, the new method is called rather than the original method. The overridden method can be invoked using the **super** variable such that:

```
setThermostat(...)         // refers to the overriding method
super.setThermostat(...)   // refers to the overridden method
```

4.4 Overload Resolution

Overloaded methods have the same name as an existing method, but differ in the number and/or the types of arguments. Overload resolution involves determining which overloaded method to invoke. The return type is not considered when resolving overloaded methods. Methods may be overloaded within the same class. The order of method declaration within a class is not significant.

Methods may be overloaded by varying both the number and the type of arguments. The compiler determines which matching method has the lowest type conversion cost. Only methods with the same name and number of arguments are considered for matching. The cost of matching a method is the maximum cost of converting any one of its arguments. There are two types of arguments to consider:, object types and base types.

The cost of converting among object types is the number of links in the class tree between the actual parameter's class and the prototype parameter's class. Only widening conversions are considered. (See "Casting Between Class Types" on page 8 for more information on object conversion.) No conversion is necessary for argument types that match exactly, making their cost 0.

The cost of converting base types is calculated from the table below. Exact matches cost 0.

		To						
		byte	short	char	int	long	float	double
From	byte	0	1	2	3	4	6	7
	short	10	0	10	1	2	4	5
	char	11	10	0	1	2	4	5
	int	12	11	11	0	1	5	4
	long	12	11	11	10	0	6	5
	float	15	14	13	12	11	0	1
	double	16	15	14	13	12	10	0

Note: Cost >= 10 causes data loss.

Once a conversion cost is assigned to each matching method, the compiler chooses the method which has the lowest conversion cost. If there is more than one potential method with the same lowest cost the match is ambiguous and a compile-time error occurs.

For example:

```
class A {
    int method(Object o, Thread t);
    int method(Thread t, Object o);

    void g(Object o, Thread t) {
        method(o, t);     // calls the first method.
        method(t, o);     // calls the second method.
        method(o, o);     // ambiguous - compile-time error
    }
}
```

Note: The names of parameters are not significant. Only the number, type, and order are.

4.5 Constructors

Constructors are special methods provided for initialization. They are distinguished by having the same name as their class and by not having any return type. Constructors are automatically called upon the creation of an object. They cannot be called explicitly through an object. If you want to be able to call the constructor outside the package, make the constructor **public** (see "Access Specifiers" on page 16 for more information).

Constructors can be overloaded by varying the number and types of parameters, just as any other method can be overloaded.

```
class Foo {
    int x;
    float y;
    Foo() {
        x = 0;
        y = 0.0;
    }
    Foo(int a) {
        x = a;
        y = 0.0;
    }
    Foo(float a) {
        x = 0;
        y = a;
    }
    Foo(int a, float b) {
        x = a;
        y = b;
    }
    static void myFoo() {
        Foo obj1 = new Foo();       //calls Foo();
        Foo obj2 = new Foo(4);      //calls Foo(int a);
        Foo obj3 = new Foo(4.0);    //calls Foo(float a);
       Foo obj4 = new Foo(4, 4.0); //calls Foo(int a, float b);
    }
}
```

The instance variables of superclasses are initialized by calling either a constructor for the immediate superclass or a constructor for the current class. If neither is specified in the code, the superclass constructor that has no parameters is invoked. If a constructor calls another constructor in this class or a constructor in the immediate super class, that call must be the first thing in the constructor body. Instance variables can't be referenced before calling the constructor.

Invoking a constructor of the immediate superclass is done as follows:

```
class MyClass extends OtherClass {
    MyClass(someParameters) {
        /* Call immediate superclass constructor */
        super(otherParameters);
        ...
    }
    ...
}
```

Invoking a constructor in the current class is done as follows:

```
class MyClass extends OtherClass {
    MyClass(someParameters) {
        ...
    }
    MyClass(otherParameters) {
        /* Call the constructor in this class that has the
           specified parameter list. */
        this(someParameters);
        ...
    }
    ...
}
```

The Foo and FooSub methods below are examples of constructors.

```
class Foo extends Bar {
    int a;
    Foo(int a) {
        // implicit call to Bar()
        this.a = a;
    }
    Foo() {
        this(42);         // calls Foo(42) instead of Bar()
    }
}

class FooSub extends Foo {
    int b;
    FooSub(int b) {
        super(13);      // calls Foo(13); without this line,
                        // would have called Foo()
        this.b = b;
    }
}
```

If a class declares no constructors, the compiler automatically generates one of the following form:

```
class MyClass extends OtherClass {
    MyClass() {    // automatically generated
        super();
    }
}
```

4.6 Object Creation—the new Operator

A class is a template used to define the state and behavior of an object. An *object* is an *instance* of a class. All instances of classes are allocated in a garbage collected heap. Declaring a reference to an object does not allocate any storage for that object. The programmer must explicitly allocate the storage for objects, but no explicit deallocation is required; the garbage collector automatically reclaims the memory when it is no longer needed.

To allocate storage for an object, use the **new** operator. In addition to allocating storage, **new** initializes the instance variables and then calls the instance's constructor. The constructor is a method that initializes an object (see "Constructors" on page 12). The following syntax allocates and initializes a new instance of a class named ClassA:

```
a = new ClassA();
```

This constructor syntax provides arguments to the constructor:

```
b = new ClassA(3,2)
```

4.6.1 *Garbage Collection*

The garbage collector makes most aspects of storage management simple and robust. Programs never need to explicitly free storage: it is done for them automatically. The garbage collector never frees pieces of memory that are still referenced, and it always frees pieces that are not. This makes both dangling pointer bugs and storage leaks impossible. It also frees designers from having to figure out which parts of a system have to be responsible for managing storage.

4.6.2 *Finalization*

The Java language includes the concept of object finalization. Java finalization is generalization of garbage collection that allows a program to free arbitrary resources (e.g., file descriptors or graphics contexts) owned by objects that cannot be accessed by any Java program. Reclaiming an object's memory by garbage collection does not guarantee that these resources will be reclaimed[1].

4.6.3 *The* **null** *Reference*

The keyword **null** is a predefined constant that represents "no instance." **null** can be used anywhere an instance is expected and can be cast to any class type.

4.7 Static Methods, Variables, and Initializers

Variables and methods declared in a class can be declared **static**, which makes them apply to the class itself, rather than to an instance of the class. In addition, a block of code within a class definition can be declared static. Such a block of code is called a static initializer.

1. When a user defines the `void finalize()` method in a class definition, finalization is enabled for objects of that class. Finalization of an object consists of the system calling the object's `finalize()` method. Finalization normally occurs asynchronously at some time after the garbage collection mechanism identifies an object as inaccessible. Users can invoke their `finalize()` method explicitly but this doesn't guarantee that the system will not call it again at a later time. If a finalized object references another finalized object, the objects are finalized in the reverse order of their creation. Java does not guarantee when or if a given finalized object will have its `finalize()` method called. Thus, finalization should not be relied on for program correctness. Rather, finalization should be thought of as an optimization.

Static variables can have initializers, just as instance variables can. See "Order of Initialization" on page 15 for more information. A static variable exists only once per class, no matter how many instances of the class exist. Both static variables and static methods are accessed using the class name. For convenience, they can also be accessed using an instance of the class.

```
class Ahem {
    int i;                              // Instance variable
    static int j;                       // Static variable
    static int arr[] = new int[12];
    static {                            // static initializer:
                                    // initialize the array
        for (int i = 0; i < arr.length; i++) {
            arr[i] = i;
        }
    }
    void seti(int i) {                  // Instance method
        this.i = i;
    }
    static void setj(int j) {           // Static method
        Ahem.j = j;
}
    static void clearThroat() {
        Ahem a = new Ahem();
        Ahem.j = 2;       // valid; static var via class
        a.j = 3;          // valid; static var via instance
        Ahem.setj(2);     // valid; static method via class
        a.setj(3);        // valid; static method via instance
        a.i = 4;          // valid; instance var via instance
        Ahem.i = 5;       // ERROR; instance var via class
       a.seti(4);        // valid; instance method via instance
        Ahem.seti(5);     // ERROR; instance method via class
    }
}
```

4.7.1 *Order of Declarations*

The order of declaration of classes and the methods and instance variables within them is irrelevant. However, it is possible for cycles to exist during initialization. For information on cycles during initialization see "Order of Initialization" on page 15. Methods are free to make forward references to other methods and instance variables. The following is legal:

```
class A {
    void a() {
        f.set(42);
    }
    B f;
}
class B {
    void set(long n) {
        this.n = n; }
    long n;
}
```

4.7.2 *Order of Initialization*

When a class is loaded, all of its static initialization code is executed. Static initializers are executed at the same time that static variables are initialized. The initializations occur in lexical order. For example, a class C is declared as follows:

```
class C {
    static int a = 1;
    static {
        a++;
```

```
        b = 7;
    }
    static int b = 2;
}
```

When class C is loaded, the following occurs in order:

- a is set to 1
- the static initializer is executed, setting a to 2 and b to 7
- b is set to 2

If any static initialization code has a reference to some other, unloaded class, that class is loaded and its static initialization code is executed first. Each unloaded class referenced during static initialization is loaded and initialized before the class that referenced it. If at any time during this initialization sequence a reference is made to an uninitialized class that is earlier in the sequence, a cycle is created. A cycle causes a NoClassDefFoundException to be thrown.

For example, if ClassA is loaded, its static initialization code is executed. However, ClassA's static initialization code can have a reference to another unloaded class, for example, ClassB. In that case, ClassB is loaded and its static initialization occurs before ClassA's. Then, ClassA's static initializations are executed. A cycle is created if ClassB has a reference to ClassA in its static initialization code.

It is an compile-time error for instance or static variable initializations to have a forward dependency. For example, the following code:

```
int i = j + 2;
int j = 4;
```

results in a compile-time error.

An instance variable's initialization can have an apparent forward dependency on a static variable. For example in the following code fragment:

```
int i = j + 2;              // Instance variable
static int j = 4;           // Static variable
```

it appears that `i` has a forward dependency on `j`. However, `i` is initialized to 6 and `j` is initialized to 4. This initialization occurs because `j` is a static variable and is initalized before the instance variable. Thus, `j` is initialized to 4 before `i` is initialized.

Static methods cannot refer to instance variables; they can only use static variables and static methods.

4.8 Access Specifiers

Access specifiers are modifiers that allow programmers to control access to methods and variables. The keywords used to control access are **public**, **private**, and **protected**. Methods marked as **public** can be accessed from anywhere by anyone. Methods marked as **private** can be accessed only from within the class in which they are declared. Since private methods are not visible outside the class, they are effectively **final** and cannot be overridden (see "Final Classes, Methods, and Variables" on page 18 for more information). Moreover, you cannot override a non-private method and give it private access. The **protected** access specifier makes a variable or method accessible to subclasses, but not to any other classes.

Public access can be applied to classes, methods, and variables. Classes, methods, and variables marked as **public** can be accessed from anywhere by any other class or method. The access of a public method cannot be changed by overriding it.

Classes, methods, and variables that do not have either private or public access specified can be accessed only from within the package where they are declared (see "Packages" on page 21).

4.9 Variable Scoping Rules

Within a package, when a class is defined as a subclass of another, declarations made in the superclass are visible in the subclass. When a variable is referenced inside a method definition, the following scoping rules are used:

1. The current block is searched first, and then all enclosing blocks, up to and including the current method. This is considered the local scope.

After the local scope, the search continues in the class scope:

2. The variables of the current class are searched.
3. If the variable is not found, variables of all superclasses are searched, starting with the immediate superclass, and continuing up through class Object until the variable is found. If the variable is not found, imported classes and package names are searched. If it is not found, it is a compile-time error.

Multiple variables with the same name within the same class are not allowed and result in a compile-time error.

4.10 Modifiers

4.10.1 *Threadsafe Variables*

An instance or static variable can be marked **threadsafe** to indicate that the variable will never be changed by some other thread while one thread is using it, i.e., the variable never changes asynchronously. The purpose of marking a variable as threadsafe is to allow the compiler to perform some optimizations that may mask the occurrence of asynchronous changes. The primary optimization enabled by the use of **threadsafe** is the caching of instance variables in registers.

4.10.2 *Transient Variables*

The **transient** flag is available to the interpreter and is intended to be used for persistent objects. Variables marked **transient** are treated specially when instances of the class are written out as persistent objects.

4.10.3 *Final Classes, Methods, and Variables*

The **final** keyword is a modifier that marks a class as never having subclasses, a method as never being overridden, or a variable as having a constant value. It is a compile-time error to override a final method, subclass a final class, or change the value of a final variable. Variables marked as **final** behave like constants.

Using **final** lets the compiler perform a variety of optimizations. One such optimization is inline expansion of method bodies, which may be done for small, final methods (where the meaning of *small* is implementation dependent).

Examples of the various final declarations are:

```
class Foo {
    final int value = 3;              // final variable
    final int foo(int a, int b) {     // final method
        ...
    }
}
```

4.10.4 *Native Methods*

Methods marked as **native** are implemented in a platform-dependent language, e.g., C, not Java Native methods do not have a method body, instead the declaration is terminated with a semicolon. Constructors cannot be marked as native. Though implemented in a platform-dependent language, native methods behave exactly as non-native methods do, for example, it is possible to override them. An example of a native method declaration is:

```
native long timeOfDay();
```

4.10.5 *Abstract Methods*

Abstract methods provide the means for a superclass or interface to define a protocol that subclasses must implement. Methods marked as abstract must be defined in a subclass of the class in which they are declared. An abstract method does not have a method body; instead the declaration is terminated with a semi-colon.

The following rules apply to the use of the **abstract** keyword:

- Constructors cannot be marked as abstract.
- Static methods cannot be abstract.
- Private methods cannot be abstract.
- Abstract methods must be defined in some subclass of the class in which they are declared.
- A method that overrides a superclass method cannot be abstract.
- Classes that contains abstract methods and classes that inherit abstract methods without overriding them are considered abstract classes.
- It is a compile-time error to instantiate an abstract class or attempt to call an abstract method directly.

4.10.6 *Synchronized Methods and Blocks*

The **synchronized** keyword is a modifier that marks a method or block of code as being required to acquire a lock. The lock is necessary so that the synchronized code does not run at the same time as other code that needs access to the same resource. Each object has exactly one lock associated with it; each class also has exactly one lock. Synchronized methods are reentrant.

When a synchronized method is invoked, it waits until it can acquire the lock for the current instance (or class, if it's a static method). After acquiring the lock, it executes its code and then releases the lock.

Synchronized blocks of code behave similarly to synchronized methods. The difference is that instead of using the lock for the current instance or class, they use the lock associated with the object or class specified in the block's **synchronized** statement.

Synchronized blocks are declared as follows:

```
/* ...preceding code in the method... */
synchronized(<object or class name>) {      //sync. block
    /* code that requires synchronized access */
}
/* ...remaining code in the method... */
```

An example of the declaration of a synchronized method is:

```
class Point {
    float x, y;
    synchronized void scale(float f) {
        x *= f;
        y *= f;
    }
}
```

An example of a synchronized block is:

```
class Rectangle {
    Point topLeft;
    ...
    void print() {
        synchronized (topLeft) {
            println("topLeft.x = " + topLeft.x);
            println("topLeft.y = " + topLeft.y);
        }
        ...
    }
}
```

5 *Interfaces*

An interface specifies a collection of methods without implementing their bodies. Interfaces provide encapsulation of method protocols without restricting the implementation to one inheritance tree. When a class implements an interface, it generally must implement the bodies of all the methods described in the interface. (If the implementing class is abstract—never implemented—it can leave the implementation of some or all of the interface methods to its subclasses.)

Interfaces solve some of the same problems that multiple inheritance does without as much overhead at runtime. However, because interfaces involve dynamic method binding, there is a small performance penalty to using them.

Using interfaces allows several classes to share a programming interface without having to be fully aware of each other's implementation. The following example shows an interface declaration (with the **interface** keyword) and a class that implements the interface:

```
public interface Storing {
    void freezeDry(Stream s);
    void reconstitute(Stream s);
}
public class Image implements Storing, Painting {
    ...
    void freezeDry(Stream s) {
        // JPEG compress image before storing
        ...
    }
    void reconstitute (Stream s) {
        // JPEG decompress image before reading
        ...
    }
}
```

Like classes, interfaces are either private (the default) or public. The scope of public and private interfaces is the same as that of public and private classes, respectively. Methods in an interface are always **public**. Variables are **public**, **static**, and **final**.

5.1 Interfaces as Types

The declaration syntax *interfaceName variableName* declares a variable or parameter to be an instance of some class that implements *interfaceName*. Interfaces behave exactly as classes when used as a type. This lets the programmer specify that an object must implement a given interface, without having to know the exact type or inheritance of that object. Using interfaces makes it unnecessary to force related classes to share a common abstract superclass or to add methods to Object.

The following pseudocode illustrates the *interfaceName variableName* syntax:

```
class StorageManager {
    Stream stream;
    ...
    // Storing is the interface name
    void pickle(Storing obj) {
        obj.freezeDry(stream);
    }
}
```

5.2 Methods in Interfaces

Methods in interfaces are declared as follows:

returnType methodName (parameterList)`;`

The declaration contains no modifiers. All methods specified in an interface are public and abstract and no other modifiers may be applied.

See “Abstract Methods” on page 18 for more information on abstract methods.

5.3 Variables in Interfaces

Variables declared in interfaces are **final**, **public**, and **static**. No modifiers can be applied. Variables in interfaces must be initialized.

5.4 Combining Interfaces

Interfaces can incorporate one or more other interfaces, using the **extends** keyword as follows:

```
interface DoesItAll extends Storing, Painting {
    void doesSomethingElse();
}
```

6 *Packages*

Packages are groups of classes and interfaces. They are a tool for managing a large namespace and avoiding conflicts. Every class and interface name is contained in some package. By convention, package names consist of period-separated words, with the first name representing the organization that developed the package.

6.1 Specifying a Compilation Unit’s Package

The package that a compilation unit is in is specified by a **package** statement. When this statement is present, it must be the first non-comment, non-white space line in the compilation unit. It has the following format:

```
package packageName;
```

When a compilation unit has no **package** statement, the unit is placed in a default package, which has no name.

6.2 Using Classes and Interfaces from Other Packages

The language provides a mechanism for making the definitions and implementations of classes and interfaces available across packages. The **import** keyword is used to mark classes as being imported into the current package. A compilation unit automatically imports every class and interface in its own package.

Code in one package can specify classes or interfaces from another package in one of two ways:

- By prefacing each reference to the class or interface name with the name of its package:

  ```
  // prefacing with a package
  acme.project.FooBar obj = new acme.project.FooBar();
  ```

- By importing the class or interface or the package that contains it, using an **import** statement. Importing a class or interface makes the name of the class or interface available in the current namespace. Importing a package makes the names of all of its public classes and interfaces available. The construct:

  ```
  // import all classes from acme.project
  import acme.project.*;
  ```

  means that every public class from acme.project is imported.

  The following construct imports a single class, `Employee_List`, from the `acme.project` package:

  ```
  // import Employee_List from acme.project
  import acme.project.FooBar;
  Employee_List obj = new Employee_List();
  ```

It is illegal to specify an ambiguous class name and doing so always generates a compile-time error. Class names may be disambiguated through the use of a fully qualified class name, i.e., one that includes the name of the class's package.

7 *Expressions*

Expressions in the language are much like expressions in C.

7.1 Operators

The operators, from highest to lowest precedence, are:

```
. [] ()
++ -- ! ~ instanceof
* / %
+ -
<< >> >>>
< > <= >=
== !=
&
^
|
&&
||
?:
= op=
,
```

7.1.1 *Operators on Integers*

For operators with integer results, if any operand is **long**, the result type is **long**. Otherwise the result type is **int**—never **byte**, **short**, or **char**. Thus, if a variable `i` is declared a **short** or a **byte**, `i+1` would be an **int**. When a result outside an operator's range would be produced, the result is reduced modulo the range of the result type.

The unary integer operators are:

Operator	Operation
−	unary negation
~	bitwise complement
++	Increment
−−	Decrement

The ++ operator is used to express incrementing directly. Incrementing can also be expressed indirectly using addition and assignment. ++lvalue means *lvalue*`+=1`. ++*lvalue* also means *lvalue=lvalue*`+1` (as long as lvalue has no side effects). The -- operator is used to express decrementing. The ++ and −− operators can be used as both prefix and postfix operators.

The binary integer operators are:

Operator	Operation[a]
+	addition
−	subtraction
*	multiplication
/	division
%	modulus
&	bitwise AND
|	bitwise OR
^	bitwise XOR
<<	left shift
>>	sign-propagating right shift
>>>	zero-fill right shift

a. **integer** *op* **integer** => **integer**

Integer division rounds toward zero. Division and modulus obey the identity `(a/b)*b + (a%b) == a`. `%0` throws an ArithmeticException.

The only exception for integer arithmetic is caused by a divide by zero, which throws the ArithmeticException. An underflow generates zero. An overflow leads to wrap-around, i.e., adding 1 to the maximum integer wraps around to the minimum integer.

An *op*= assignment operator corresponds to each of the binary operators in the above table.

The integer relational operators **<**, **>**, **<=**, **>=**, **==**, and **!=** produce **boolean** results.

The operators `abs(x)`, `max(x,y)`, and `min(x,y)` work for integers as they do for all numbers.

7.1.2 *Operators on Boolean Values*

Variables or expressions that are **boolean** can be combined to yield other **boolean** values. The unary operator **!** is boolean negation. The binary operators **&**, **|**, and **^** are the logical AND, OR, and XOR operators; they force evaluation of both operands. To avoid evaluation of right-hand operands, you can use the short-cut evaluation operators **&&** and **||**. You can also use == and !=. The assignment operators also work: **&=**, **|=**, **^=**. The ternary conditional operator ?: works as it does in C.

7.1.3 *Operators on Floating Point Values*

Floating point values can be combined using the usual operators: unary **–**; binary **+**, **–**, *, and **/**; and the assignment operators **+=**, **–=**, *=, and **/=**. The ++ and -- operators also work on floating point values (they add or subtract 1.0). In addition, % and %= work on floating point values, i.e.,

```
a % b
```

is the same as:

```
a - ((int)(a / b) * b)
```

This means that a%b is the floating point equivalent of the remainder after division.

The operators `abs(x)`, `max(x,y)`, and `min(x,y)` work for floats as they do for all numbers.

Floating point expressions involving only single-precision operands are evaluated using single-precision operations and produce single-precision results. Floating point expressions that involve at least one double-precision operand are evaluated using double-precision operations and produce double-precision results.

The language has no arithmetic exceptions for floating point arithmetic. Following the IEEE 754 floating point specification, the distinguished values Inf and NaN are used instead. Overflow generates Inf. Underflow generates 0. Divide by zero generates Inf.

The usual relational operators are also available and produce **boolean** results: **>**, **<**, **>=**, **<=**, **==**, **!=**. Because of the properties of NaN, floating point values are not fully ordered, so care must be taken in comparison. For instance, if `a<b` is not true, it does not follow that `a>=b`. Likewise, `a!=b` does not imply that `a>b || a<b`. In fact, there may no ordering at all.

Floating point arithmetic and data formats are defined by IEEE 754, "Standard for Floating Point Arithmetic." See "Appendix: Floating Point" on page 33 for details on the language's floating point implementation.

7.1.4 *Operators on Arrays*

The following:

```
<expression>[<expression>]
```

gets the value of an element of an array. Legal ranges for the expression are from 0 to the length of the array minus 1. The range is checked only at runtime.

7.1.5 *Operators on Strings*

Strings are implemented as String objects (see "String Literals" on page 4 for more information). The operator + concatenates Strings, automatically converting operands into Strings if necessary. If the operand is an object it can define a method call `toString()` that returns a String in the class of the object.

```
// Examples of the + operator used with strings
float a = 1.0;
print("The value of a is " + a + "\n");
String s = "a = " + a;
```

The += operator works on Strings. Note, that the left hand side (`s1` in the following example) is evaluated only once.

```
s1 += a; //s1 = s1 + a; // a is converted to String if necessary
```

7.1.6 *Operators on Objects*

The binary operator **instanceof** tests whether the specified object is an instance of the specified class or one of its subclasses. For example:

```
if (thermostat instanceof MeasuringDevice) {
    MeasuringDevice dev = (MeasuringDevice)thermostat;
       ...
}
```

determines whether thermostat is a MeasuringDevice object (an instance of MeasuringDevice or one of its subclasses).

7.2 Casts and Conversions

The Java language and runtime system restrict casts and conversions to help prevent the possibility of corrupting the system. Integers and floating point numbers can be cast back and forth, but integers cannot be cast to arrays or objects. Objects cannot be cast to base types. An instance can be cast to a superclass with no penalty, but casting to a subclass generates a runtime check. If the object being cast to a subclass is not an instance of the subclass (or one of its subclasses), the runtime system throws a ClassCastException.

8 *Statements*

8.1 Declarations

Declarations can appear anywhere that a statement is allowed. The scope of the declaration ends at the end of the enclosing block.

In addition, declarations are allowed at the head of **for** statements, as shown below:

```
for (int i = 0; i < 10; i++) {
    ...
}
```

Items declared in this way are valid only within the scope of the **for** statement. For example, the preceding code sample is equivalent to the following:

```
{
    int i = 0;
    for (; i < 10; i++) {
    ...
    }
}
```

8.2 Expressions

Expressions are statements:

```
a = 3;
print(23);
foo.bar();
```

8.3 Control Flow

The following is a summary of control flow:

```
if(boolean) statement
else statement
switch(e1) {
          case e2: statements
          default: statements
}
break [label];
continue [label];
return e1;
for([e1]; [e2]; [e3]) statement
while(boolean) statement
do statement while(boolean);
label:statement
```

The language supports labeled loops and labeled breaks, for example:

```
outer:   // the label
    for (int i = 0; i < 10; i++) {
        for (int j= 0; j< 10; j++) {
            if (...) {
                break outer;
            }
            if (...) {
            }
        }
}
```

The use of labels in loops and breaks has the following rules:

- Any statement can have a label.
- If a break statement has a label it must be the label of an enclosing statement.
- If a continue statement has a label it must be the label of an enclosing loop.

8.4 Exceptions

When an error occurs in an Java program—for example, when an argument has an invalid value—the code that detects the error can *throw* an exception[1]. By default, exceptions result in the thread terminating after printing an error message. However, programs can have *exception handlers* that *catch* the exception and recover from the error.

Some exceptions are thrown by the Java runtime system. However, any class can define its own exceptions and cause them to occur using **throw** statements. A **throw** statement consists of the **throw** keyword followed by an object. By convention, the object should be an instance of Exception or one of its subclasses. The **throw** statement causes execution to switch to the appropriate exception handler. When a **throw** statement is executed, any code following it is not executed, and no value is returned by its enclosing method. The following example shows how to create a subclass of Exception and throw an exception.

```
class MyException extends Exception {
}

class MyClass {
    void oops() {
        if (/* no error occurred */) {
            ...
        } else { /* error occurred */
        throw new MyException();
        }
    }
}
```

To define an exception handler, the program must first surround the code that can cause the exception with a **try** statement. After the **try** statement come one or more **catch** statements—one per exception class that the program can handle at that point. In each **catch** statement is exception handling code. For example:

```
try {
    p.a = 10;
} catch (NullPointerException e) {
    println("p was null");
} catch (Exception e) {
    println("other error occurred");
} catch (Object obj) {
    println("Who threw that object?");
}
```

A **catch** statement is like a method definition with exactly one parameter and no return type. The parameter can be either a class or an interface. When an exception occurs, the nested **try/catch** statements are searched for a parameter that matches the exception class. The parameter is said to match the exception if it:

- is the same class as the exception; or
- is a superclass of the exception; or
- if the parameter is an interface, the exception class implements the interface.

1. Java exception handling closely follows the proposal in the second edition of *The C++ Programming Language*, by Bjarne Stroustrup.

The first **try/catch** statement that has a parameter that matches the exception has its **catch** statement executed. After the **catch** statement executes, execution resumes after the **try**/**catch** statement. It is not possible for an exception handler to resume execution at the point that the exception occurred. For example, this code fragment:

```
print("now ");
try {
    print("is ");
    throw new MyException();
    print("a ");
} catch(MyException e) {
    print("the ");
}
print("time\n");
```

prints "now is the time". As this example shows, exceptions don't have to be used only for error handling, but any other use is likely to result in code that's hard to understand.

Exception handlers can be nested, allowing exception handling to happen in more than one place. Nested exception handling is often used when the first handler can't recover completely from the error, yet needs to execute some cleanup code (as shown in the following code example). To pass exception handling up to the next higher handler, use the **throw** keyword using the same object that was caught. Note that the method that rethrows the exception stops executing after the **throw** statement; it never returns.

```
try {
    f.open();
} catch(Exception e) {
    f.close();
    throw e;
}
```

8.4.1 *The* **finally** *Statement*

The following example shows the use of a **finally** statement that is useful for guaranteeing that some code gets executed whether or not an exception occurs. You can use either a **catch** statement or a **finally** statement within a particular **try** block, but not both. For example, the following code example:

```
try {
    // do something
} finally {
    // clean up after it
}
```

is similar to:

```
try {
    // do something
} catch(Object e){
    // clean up after it
    throw e;
}
// clean up after it
```

The **finally** statement is executed even if the **try** block contains a **return**, **break**, **continue**, or **throw** statement. For example, the following code example always results in "finally" being printed, but "after try" is printed only if a != 10.

```
try {
```

```
        if (a == 10) {
            return;
        }
    } finally {
        print("finally\n");
    }
    print("after try\n");
```

8.4.2 *Runtime Exceptions*

This section contains a list of the exceptions that the Java runtime throws when it encounters various errors.

ArithmeticException

Attempting to divide an integer by zero throws the ArithmeticException—no other arithmetic operation in Java throws an exception. For information on how Java handles other arithmetic errors see “Operators on Integers” on page 22 and “Operators on Floating Point Values” on page 24.

For example, the following code causes an ArithmeticException to be thrown:

```
class Arith {
    public static void main(String args[]) {
        int j = 0;
        j = j / j;
    }
}
```

NullPointerException

An attempt to access a variable or method in a null object or a element in a null array throws a NullPointerException. For example, the accesses o.length and a[0] in the following class declaration throws a NullPointerException at runtime.

```
class Null {
    public static void main(String args[]) {
        String o = null;
        int a[] = null;
        o.length();
        a[0] = 0;
    }
}
```

It is interesting to note that if you throw a null object you actually throw a NullPointerException.

IncompatibleClassChangeException

In general the IncompatibleClassChangeException is thrown whenever one class’s definition changes but other classes that reference the first class aren’t recompiled. Four specific changes that throw a IncompatibleClassChangeException at rutime are:

- A variable’s declaration is changed from static to non-static in one class but other classes that access the changed variable aren’t recompiled.
- A variable’s declaration is changed from non-static to static in one class but other classes that access the changed variable aren’t recompiled.
- A field that is declared in one class is deleted but other classes that access the field aren’t recompiled.

- A method that is declared in one class is deleted but other classes that access the method aren't recompiled.

ClassCastException

A ClassCastException is thrown if an attempt is made to cast an object O into a class C and O is neither C nor a subclass of C. For more information on casting see "Casting Between Class Types" on page 8.

The following class declaration results in a ClassCastException at runtime:

```
class ClassCast {
    public static void main(String args[]) {
        Object o = new Object();
        String s = (String)o;           // the cast attempt
        s.length();
    }
}
```

NegativeArraySizeException

A NegativeArraySizeException is thrown if an array is created with a negative size. For example, the following class definition throws a NegativeArraySizeException at runtime:

```
class NegArray {
    public static void main(String args[]) {
        int a[] = new int[-1];
        a[0] = 0;
    }
}
```

OutOfMemoryException

An OutOfMemoryException is thrown when the system can no longer suppy the application with memory. The OutOfMemoryException can only occur during the creation of an object, i.e., when **new** is called. For example, the following code results in an OutOfMemoryException at runtime:

```
class Link {
    int a[] = new int[1000000];
    Link l;
}
class OutOfMem {
    public static void main(String args[]) {
        Link root = new Link();
        Link cur = root;
        while(true) {
            cur.l = new Link();
            cur = cur.l;
        }
    }
}
```

NoClassDefFoundException

A NoClassDefFoundException is thrown if a class is referenced but the runtime system cannot find the referenced class.

For example, class NoClass is declared:

```
class NoClass {
    public static void main(String args[]) {
```

```
            C c = new C();
        }
    }
```

When NoClass is run, if the runtime system can't find C.class it throws the NoClassDefFoundException.

Note: C.class must have existed at the time NoClass is compiled.

IncompatibleTypeException

An IncompatibleTypeException is thrown if an attempt is made to instantiate an interface. For example, the following code causes an IncompatibleTypeException to be thrown.

```
    interface I {
    }

    class IncompType {
        public static void main(String args[]) {
            I r = (I)new("I");
        }
    }
```

ArrayIndexOutOfBoundsException

An attempt to access an invalid element in an array throws an ArrayIndexOutOfBoundsException. For example:

```
    class ArrayOut {
        public static void main(String args[]) {
            int a[] = new int[0];
            a[0] = 0;
        }
    }
```

UnsatisfiedLinkException

An UnsatisfiedLinkException is thrown if a method is declared native and the method cannot be linked to a routine in the runtime.

```
    class NoLink {
        static native void foo();

        public static void main(String args[]) {
            foo();
        }
    }
```

InternalException

An InternalException should never be thrown. It's only thrown if some consistency check in the runtime fails. Please send mail to WebRunner@Sun.COM if you have a reproducible case that throws this exception.

A Appendix: Floating Point

This appendix discusses properties of Java floating point arithmetic: general precision notes and special values, binary format conversion, ordering. At the end is a section summarizing the differences between Java arithmetic and the IEEE 754 standard. For more information on the IEEE 754 standard, see "IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std. 754-1985."

Operations involving only single-precision **float** and integer values are performed using at least single-precision arithmetic and produce a single-precision result. Other operations are performed in double precision and produce a double precision result. Java floating-point arithmetic produces no exceptions.

Underflow is gradual.

A.1 Special Values

There is both a positive zero and a negative zero. The latter can be produced in a number of special circumstances: the total underflow of a * or / of terms of different sign; the addition of -0 to itself or subtraction of positive zero from it; the square root of -0. Converting -0 to a string results in a leading '-'. Apart from this, the two zeros are indistinguishable.

Calculations which would produce a value beyond the range of the arithmetic being used deliver a signed infinite result. An infinity (`Inf`) has a larger magnitude than any value with the same sign. Infinities of the same sign cannot be distinguished. Thus, for instance `(1./0.) + (1./0.) == (1./0.)`. Division of a finite value by infinity yields a 0 result.

Calculations which cannot produce any meaningful numeric result deliver a distinguished result called Not A Number (`NaN`). Any operation having a NaN as an operand produces a NaN as the result. NaN is not signed and not ordered (see "Ordering"). Division of infinity by infinity yields NaN, as does subtraction of one infinity from another of the same sign.

A.2 Binary Format Conversion

Converting a floating-point value to an integer format results in a value with the same sign as the argument value and having the largest magnitude less than or equal to that of the argument. In other words, conversion rounds towards zero. Converting infinity or any value beyond the range of the target integer type gives a result having the same sign as the argument and the maximum magnitude of that sign. Converting NaN results in 0.

Converting an integer to a floating format results in the closest possible value in the target format. Ties are broken in favor of the most even value (having 0 as the least-significant bit).

A.3 Ordering

The usual relational operators can be applied to floating-point values. With the exception of `NaN`, all floating values are ordered, with `-Inf` < all finite values < `Inf`.

`-Inf == -Inf, +Inf == +Inf, -0. == 0`. The ordering relations are transitive. Equality and inequality are reflexive.

`NaN` is unordered. Thus the result of any order relation between NaN and any other value is false and produces 0. The one exception is that "`NaN !=` anything" is true.

Note that, because NaN is unordered, Java's logical inversion operator, !, does not distribute over floating point relationals as it can over integers.

A.4 Summary of IEEE-754 Differences

Java arithmetic is a subset of the IEEE-754 standard. Here is a summary of the key differences.

- Nonstop Arithmetic—The Java system will not throw exceptions, traps, or otherwise signal the IEEE exceptional conditions: invalid operation, division by zero, overflow, underflow, or inexact. Java has no signaling NaN.
- Rounding—Java rounds inexact results to the nearest representable value, with ties going to the value with a 0 least-significant bit. This is the IEEE default mode. But, Java rounds towards zero when converting a floating value to an integer. Java does not provide the user-selectable rounding modes for floating-point computations: up, down, or towards zero.
- Relational set—Java has no relational predicates which include the unordered condition, except for !=. However, all cases but one can be constructed by the programmer, using the existing relations and logical inversion. The exception case is ordered but unequal. There is no specific IEEE requirement here.
- Extended formats—Java does not support any extended formats, except that double will serve as single-extended. Other extended formats are not a requirement of the standard.

MIndex

Symbols

!, 24
–, 23, 24
!=, 23, 24
%, 23
&, 23, 24
&&, 24
&=, 24
*, 23, 24
*=, 24
+, 23, 24, 25
+=, 24
-, unary, 23
–, unary, 24
/, 23, 24
/=, 24
<, 23, 24
<<, 23
<=, 23, 24
–=, 24
==, 23, 24
>, 23, 24
>=, 23, 24
>>, 23
>>>, 23
^, 23, 24
^=, 24
|, 23, 24
|=, 24
||, 24
~, 23

B

boolean, 3
boolean expressions, 26
break, 26
byte, 5

C

case, 26
casting, 8, 25
catch, 27
char, 5
classes, 5, 7, 21, 25
comments, 1
constructors, 12
continue, 26

D

declaration order, 15
default, 26
do, 26
double, 5
double precision, 3, 5, 24

E

else, 26
exceptions, 27
extends, 7

F

final, 18
finally, 28
float, 5
floating point, 3, 5, 24
floating point, ordering of values, 24
for, 25, 26

I

if, 26
implements, 7
import, 22
instanceof, 25
int, 5
integers, 5, 22
interface, 20

interfaces, 7, 19

L

length
    length of an array, 6
literals, 3
long, 5

M

methods, 9

O

object storage, 14
    (See also memory management)
objects, 13
OR, logical, 24

P

package, 21
packages, 21

R

return, 26

S

short, 5
static, 14
static initializer, 14
String, 4, 25
strings, 4, 6, 25
super, 13
switch, 26
synchronize, 19
synchronized, 19

T

this, 10
throw, 27
transient, 17
try, 27

U

Unicode, 1
    characters, 5

V

void, 9

W

while, 26

X

XOR, logical, 24