

**Java Study Notes**  
**Version 1.0 (2019-02-23)**

# Java Study Notes

1	Basic Characteristics and Features of Java .....	4
2	Terms.....	10
3	Source Files / Packages / Imports.....	11
4	COMMENTS.....	13
5	DATA TYPES .....	14
6	Primitive Types .....	15
7	Literals .....	16
8	All variables are value typed .....	18
9	VARIABLES: BASIC DEFINITIONS.....	20
10	Class Declaration, Object Instantiation, and Variable Declaration.....	22
11	Kinds of Variables.....	24
12	Cast Expressions.....	25
13	Conversions and Contexts (including casting) .....	26
14	CASTING PRIMITIVES .....	29
15	Wrapped primitives.....	31
16	Primitive Boxing / Unboxing .....	34
17	CASTING OBJECTS .....	35
18	Widening Casts of Objects .....	36
19	Narrowing .....	37
20	Narrowing: Gory Details .....	38
21	Assignment Conversion Context Details .....	41
22	Casting Conversion Context Details.....	47
23	NumericConversion Context Details .....	48
24	Instanceof operator.....	50
25	Tokens, Expressions, Statement, Declarations, Blocks.....	53
26	Labels .....	55
27	If statements .....	56
28	Short ifs .....	57
29	Switch statement .....	58
30	Conditional (ie Boolean) Expressions .....	61
31	The ternary expression.....	62
32	while loop.....	64
33	do while loop.....	65
34	for loop.....	66
35	Enhanced for ("for each") loop .....	68
36	return, break, continue.....	69
37	break, continue with label .....	71
38	Class Inheritance.....	74
39	Class Constructors .....	75
40	Class inheritance examples .....	77
41	Class members.....	80
42	Access Modifiers .....	82
43	Static Fields: Access by Class Name.....	83
44	Static Fields: Accessing from Class Instances.....	84
45	Static Methods .....	85
46	Accessing static fields/methods from static methods.....	86
47	Accessing static fields/methods from instance methods.....	87
48	Instance (ie non-static) Fields .....	89
49	Instance (ie non-static) Methods .....	90
50	The definition of this .....	91
51	The exact definition of super .....	92
52	Instance Field and Method Example .....	94
53	Method Signature and Overloading.....	97
54	Static Field Initialization and Initialization Blocks .....	100
55	Instance Field Initialization and Initialization Blocks.....	104
56	Local Variables .....	109
57	Method Parameters and Arguments .....	112
58	varargs.....	113

59	Uses of final Keyword .....	114
60	Uses of abstract Keyword .....	116
61	Uses of default keyword .....	117
62	keyword native .....	118
63	keyword transient .....	120
64	keyword volatile .....	122
65	Arrays .....	125
66	Interfaces.....	129
67	Enumerations .....	131
68	Generics .....	135
69	Generics Example Zoo .....	144
70	Reference Types Definitions .....	146
71	Heap pollution (4.12.2) .....	147
72	Type erasure .....	149
73	Reifiable Types.....	152
74	REFLECTION (RUN-TIME INFORMATION) .....	153
75	Hacks.....	155
76	CONCURRENCY (Multi-Threading).....	156
77	COLLECTIONS .....	157
78	Lambda expressions .....	158
79	Modules .....	159
80	JAVADOC.....	160

# 1 Basic Characteristics and Features of Java

## 1.1 Java

- object-oriented
- portable and architecture independent - compiles to bytecode and runs on virtual machine
- uses garbage collection and no pointers (as opposed to C++)
- does not allow bit-level control like C and C++ (size of a boolean is not defined by spec)
- distributed: integrated support for TCP/IP and Web protocols
- extensive class libraries for data structures, algorithms, I/O, and GUI
- dynamically linked - at .class file level
- supports exception handling (try/catch)
- supports multi-threading
- GUI support: AWT (Abstract Window Toolkit) / Swing (Java SE 2)
- Reflection
- JavaBeans: encapsulate many objects into one, but they are mutable

## 1.2 Java SE 5

- support for functions with variable arguments like C/C++
- generics support
- annotations (metadata and compiler directives)
- autoboxing
- concurrency utilities
- for each style looping
- enumerations
- static import

## 1.3 Java SE 6

- Improved Web Service support through [JAX-WS](#) ([JSR 224](#)).
- Java Compiler API ([JSR 199](#)): an API allowing a Java program to select and invoke a Java Compiler programmatically. That is, your code can create code and load it in run-time.

## 1.4 Java SE 7

- parallel programming / multi-tasking (Fork/Join like Linux)
- can use strings in switch statements
- binary literals
- type inference for generics
- Allowing underscores in numeric literals[\[152\]](#)
- Catching multiple exception types and rethrowing exceptions with improved type checking

## 1.5 Java SE 8

- functional programming support (lambda functions)
- functional interface support `java.util.function`

- interfaces with method definitions (support state-less, compile-time mixins)
- new stream API `java.util.stream`
- JavaFX fully integrated
- `java.util.Optional` class to avoid null reference exceptions

## 1.6 Java SE 9

- modules and `.jmod` files
- Java applet API deprecated and no more JREs, only JDK
- `JShell` command line
- `@index` annotation to the `javadoc` tool

## 1.7 Java SE 10

- local type inference via **var** (Java SE 10)
- Java version string formatting changed

## 1.8 Java SE 11

- `var` made available for lambda functions
- Applet support removed
- JavaFX removed from the JDK and is now a separate open source project
- [Java EE](#) and [CORBA](#) modules removed from JDK
- improved http support `java.net.http`

## 1.9 Java EE Libraries

- REST (Representational State Transfer)

## 1.10 3rd Party Java Libraries

- [swingLabs](#) – Extensions to Swing that might be included in Swing in the future.
- [Standard Widget Toolkit](#) – A third party widget toolkit maintained by the [Eclipse Foundation](#).
- JavaFX (separate as of Java 11)
- Common Object Request Broker Architecture (CORBA)

## 1.11 Languages that also run on the JVM

Apart from the [Java language](#), the most common or well-known other JVM languages are:

- [Clojure](#), a modern, [dynamic](#), and [functional dialect](#) of the [Lisp programming language](#)
- [Groovy](#), a dynamic programming and [scripting language](#)
- [JRuby](#), an implementation of [Ruby](#)
- [Jython](#), an implementation of [Python](#)
- [Rhino](#) – A [JavaScript](#) interpreter
- [Kotlin](#), a statically-typed language from [JetBrains](#), the developers of [IntelliJ IDEA](#)
- [Scala](#), a [statically-typed object-oriented](#) and [functional programming](#) language<sup>[1]</sup>
- [Gosu](#) – A general-purpose Java Virtual Machine-based programming language released under the Apache

- [BeanShell](#) – A lightweight scripting language for Java

## 1.12 Java Files

**java** = source file, called a **compilation unit**

may contain one or more classes

**class** = compiled java file, ready for deployment on any machine that runs the JVM, consists of bytecode

**jar** = java archive file = java zip file: its compressed and contains one or more files, packages, images etc

**war** = web archive = html, jsp

**ear** = java enterprise file for Java EE

**jmod** = java module

## 1.13 executable via main

To create a Java executable, one and only one of the classes in the .class file must have a static method named main, similar to C/C++, with the following signature.

```
public static void main(String args[] )
```

Moreover, the file must have the same name as the class that contains the main method.

## 1.14 compiling at command line

**HelloWorld.java**

```
class HelloWorld {  
    public static void main (String args[]) {  
        System.out.println("Hello World!");  
    }  
}
```

The compiled bytecode is stored in **HelloWorld.class**. This can be executed as follows:

```
> javac HelloWorld
```

In other words, the compiled file is not a native executable for the o/s. It is a native executable for the JVM.

## 1.15 compiling via IDEs

The IDE compiles automatically after each save. Each project is a Java application.

## 1.16 Java Development Kit (JDK), Java Runtime Environment (JRE), Java Virtual Machine (JVM)

### 1.16.1 JDK

Java Development Kit is the core component of Java Environment and provides all the tools, executables and binaries required to compile, debug and execute a Java Program. JDK is a platform specific software, ie the java executable for a given operating system, and that's why we have separate installers for Windows (`java.exe`), Mac and Unix (`java`) systems. We can say that JDK is superset of JRE since it contains JRE with Java compiler, debugger and core classes. Includes source for the libraries.

### 1.16.2 JRE

JRE contains the implementation of JVM, it provides platform to execute java programs. JRE consists of JVM and java binaries and other classes to execute any program successfully. JRE doesn't contain any development tools like java compiler, debugger etc. If you want to execute any java program, you should have JRE installed but we don't need JDK for running any java program.

### 1.16.3 JVM

JVM is the heart of java programming language. When we run a program, JVM is responsible to converting bytecode to the machine specific code. JVM is also platform dependent and provides core java functions like memory management, garbage collection, security etc. JVM is customizable and we can use java options to customize it, for example allocating minimum and maximum memory to JVM. JVM is called virtual because it provides a interface that does not depend on the underlying operating system and machine hardware. This independence from hardware and operating system is what makes java program write-once run-anywhere.

#### Source compilation and bytecode

The `javac` compiler compiles `.java` source files (text) into `.class` bytecode files (binary). The bytecode can be disassembled (using `javap` from the command line) to reproduce a `.java` source text file. Albeit, the disassembled file will be slightly different from the original. Bytecode is basically JVM assembly language (represented in binary form) for the JVM. It includes symbols and enough metadata to be disassembled into source.

The JDK/JRE does not run java source. It runs java bytecode.

For example, compile the following class declaration:

#### `DocFooter.java`

```
import java.awt.*;
import java.applet.*;

public class DocFooter extends Applet {
    String date;
    String email;

    public void init() {
        resize(500,100);
        date = getParameter("LAST_UPDATED");
        email = getParameter("EMAIL");
    }

    public void paint(Graphics g) {
        g.drawString(date + " by ",100, 15);
        g.drawString(email,290,15);
    }
}
```

The output from `javap DocFooter.class` yields:

```
Compiled from "DocFooter.java"
public class DocFooter extends java.applet.Applet {
    java.lang.String date;
```

```

java.lang.String email;
public DocFooter();
public void init();
public void paint(java.awt.Graphics);
}

```

The output from `javap -c DocFooter.class` yields:

```

Compiled from "DocFooter.java"
public class DocFooter extends java.applet.Applet {
    java.lang.String date;

    java.lang.String email;

    public DocFooter();
        Code:
            0: aload_0
            1: invokespecial #1                // Method
java/applet/Applet."<init>":()V
            4: return

    public void init();
        Code:
            0: aload_0
            1: sipush        500
            4: bipush        100
            6: invokevirtual #2                // Method resize:(II)V
            9: aload_0
           10: aload_0
           11: ldc           #3                // String LAST_UPDATED
           13: invokevirtual #4                // Method
getParameter:(Ljava/lang/String;)Ljava/lang/String;
           16: putfield     #5                // Field date:Ljava/lang/String;
           19: aload_0
           20: aload_0
           21: ldc           #6                // String EMAIL
           23: invokevirtual #4                // Method
getParameter:(Ljava/lang/String;)Ljava/lang/String;
           26: putfield     #7                // Field email:Ljava/lang/String;
           29: return

    public void paint(java.awt.Graphics);
        Code:
            0: aload_1
            1: new           #8                // class java/lang/StringBuilder
            4: dup
            5: invokespecial #9                // Method
java/lang/StringBuilder."<init>":()V
            8: aload_0
            9: getfield     #5                // Field date:Ljava/lang/String;
           12: invokevirtual #10               // Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
           15: ldc           #11               // String by
           17: invokevirtual #10               // Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
           20: invokevirtual #12               // Method
java/lang/StringBuilder.toString:()Ljava/lang/String;
           23: bipush        100
           25: bipush        15
           27: invokevirtual #13               // Method
java/awt/Graphics.drawString:(Ljava/lang/String;II)V

```



```
30: aload_1
31: aload_0
32: getfield      #7           // Field email:Ljava/lang/String;
35: sipush       290
38: bipush       15
40: invokevirtual #13         // Method
java/awt/Graphics.drawString:(Ljava/lang/String;II)V
43: return
}
```

## Just-in-time Compiler (JIT)

JIT is part of JVM that compiles bytecode to machine specific language (ie CPU instruction set and native o/s systems calls). The JIT performs optimization by compiling similar bytecodes at same time, hence reducing overall time taken for compilation of bytecode to machine specific language.

JRE (originally from Sun, now from Oracle) features two virtual machines, one called Client and the other Server. The Client version is tuned for quick loading. It makes use of interpretation. The Server version loads more slowly, putting more effort into producing highly optimized JIT compilations to yield higher performance. Both VMs compile only often-run methods, using a configurable invocation-count threshold to decide which methods to compile.

*Tiered compiling*, an option introduced in Java 7, uses both the client and server compilers in tandem to provide faster startup time than the server compiler, but similar or better peak performance. Starting in Java 8, tiered compilation is the default for the server VM.

There are several implementations of the JIT, including HotSpot (Oracle) and OpenJDK HotSpot (Open source).

## HotSpot

HotSpot is the proprietary JVM owned by Oracle. HotSpot is *written in C++*. In 2007, Sun estimated it comprised approximately 250,000 lines of source code. Hotspot provides:

- A Java Classloader
- A Java bytecode interpreter
- Client and Server virtual machines, optimized for their respective uses
- Several garbage collectors
- A set of supporting runtime libraries

## 2 Terms

## 3 Source Files / Packages / Imports

### 3.1 No code (other than package and import) is allowed outside of class/interface/enum definitions (in contrast to C++)

Imports are always placed outside a class and after the package statement.

An **ordinary compilation unit** [ie a .java file] consists of three parts, each of which is optional:

- A **package declaration** (§7.4), giving the fully qualified name (§6.7) of the package to which the compilation unit belongs.
  - A compilation unit that has no **package declaration** is part of an unnamed package (§7.4.2).
- **import declarations** (§7.5) that allow types from other packages and **static** members of types to be referred to using their simple names.
- Top level **type declarations** (§7.6) of **class** [including **enum**] and **interface** types.

### 3.2 Java source file example

```
// 1. *package declaration* must be first
package ron.examples;

// 2. import declarations must be second
import static java.lang.System.out; // imports the static member 'out' of class 'System'
import static java.lang.Math.*;     // imports all static members of class Math
import java.util.HashSet; // imports class HashSet so that it can be used

// 3. The type declarations must be third.
// The declarations themselves can be in any order.
// That is, in contrast to c/c++, we don't need to define types or functions
// above the places where they are used.

// Class declaration for HelloWorld.
// Thus HelloWorld is a class type or simply a "class"
public class HelloWorld implements Named {

    public static void main(String[] args) {

        int i = -25;
        int imag = Math.abs(i);
        imag = abs(imag - 100); // "import static java.lang.Math.*" allows this shorthand

        // without import, the full class name must be used
        java.util.TreeSet<Integer> mysortedNums = new java.util.TreeSet<>();
        mysortedNums.add(7);
        mysortedNums.add(1);

        // "import java.util.HashSet" allows us to use just the class name
        HashSet<Integer> myNums = new HashSet<>();
        myNums.add(7);
        myNums.add(1);
    }
}
```

```

        HelloWorld hw = new HelloWorld();
        hw.setName("Ron");
        hw.printHelloWorld();
    }

    private String name = "default";

    private SectionsOfTheDay sectionOfDay = SectionsOfTheDay.AFTERNOON;

    // @Override explicitly tell compiler and programmer that
    // this overrides an inherited method
    // That way modifications that break the signature will be noticed
    @Override
    public String getName() {
        return this.name;
    }

    @Override
    public void setName(String name) {
        this.name = name;
    }

    public void printHelloWorld() {
        System.out.println(this.getName() + " says: Hello world.");

        // "import static java.lang.System.out" allows this shorthand
        out.println(this.name + " says: Hello World!");
        out.println(this.name + " says: Good "+this.sectionOfDay);
    }
}

//Interface declaration for Named.
//Thus Named is an interface type or simply an interface
interface Named {
    String getName();
    void setName(String name);
}

//enum declaration for SectionsOfTheDay.
//Thus SectionsOfTheDay is an enum type or simply an enum
//Technically an enum is a special kind of class
enum SectionsOfTheDay {
    MORNING,
    AFTERNOON,
    NIGHT
}

// nothing is needed to terminate a .java file

```

#### output

```
<terminated> HelloWorld (1) [Java Application] C:\Program Files\Java\jdk-11.0.2\bin\javaw.exe (Feb 5, 2019, 7:55:28 PM)
```

```
Ron says: Hello world.
Ron says: Hello World!
Ron says: Good AFTERNOON
|
```

## 4 COMMENTS

Like C++, Java has two types of basic comments: multiline and rest-of-line.

```
private int test; // double slash comment

// comment // */ /* once a double slash is encountered the rest of the line is ignored

// /* hello */ once a double slash is encountered the rest of the line is ignored

/* multi-line comment
 * // double slash has no effect inside multi-line comment
 *
 *
 */
```

Java also supports a **third type of comment** for autogenerating documentation using the javadoc utility. In fact this is how the online java API documentation is created. A **javadoc comment** starts with `/**` and ends with `*/`.

## 5 DATA TYPES

### 5.1 Static Typing and Strong Typing

The Java programming language is a *statically typed* language, which means that every variable and every expression has a type that is known at compile time.

The Java programming language is also a *strongly typed* language, because types limit the values that a variable ([§4.12](#)) can hold or that an expression can produce, limit the operations supported on those values, and determine the meaning of the operations. Strong static typing helps detect errors at compile time.

### 5.2 Two data types--primitive and reference

There are two kinds of types in the Java programming language: primitive types ([§4.2](#)) and reference types ([§4.3](#)). There are, correspondingly, two kinds of data values that can be stored in variables, passed as arguments, returned by methods, and operated on: primitive values ([§4.2](#)) and reference values ([§4.3](#)).

Type :

[PrimitiveType](#)

[ReferenceType](#)

**Primitives** are not objects. they do not have constructors. they are simply stored literals (ie fixed values)

There is also a special **null type**, the type of the expression **null** ([§3.10.7](#), [§15.8.1](#)), which has no name.

Because the null type has no name, it is impossible to declare a variable of the null type or to cast to the null type.

The null reference is the only possible value of an expression of null type.

The null reference can always be assigned or cast to any reference type ([§5.2](#), [§5.3](#), [§5.5](#)).

The null reference is the unique reference to no object.

### Reference Types

An **object** is an instance of a class. Every object has a unique identifier (aka *reference* or *reference value*) that is internal to the JVM. The JVM keeps track of all extant objects in a running executable, as well as keeping track of where (ie which variables in which classes) each is used. If an object is not used anywhere, it will eventually be discarded by the garbage collector, along with identifier. The class [java.lang.Object](#) serves as the root class for all Java objects. Object are mutable and there is no inherent mechanism in java to make an object immutable. Classes are available for immutable collections.

There may be many references to the same object. Most objects have state, stored in the fields of objects that are instances of classes.. If two variables contain references to the same object, the state of the object can be modified using one variable's reference to the object, and then the altered state can be observed through the reference in the other variable.

**Arrays** are objects, although special objects and officially they are not called objects in the documentation. They have some special properties built into the Java syntax, such as use of the square brackets []. The term **component** is also used for an array element.

**Strings** are objects but have special syntax defined for string literals. Unlike all other objects they are inherently immutable.

**Classes** and **Interfaces** (officially of Class type and Interface type) are Reference Types and are referred to by a referenced value.

By convention, class, interface, and enum names should start with a capital letter. Going against this convention generates a compile-time warning.

## 6 Primitive Types

The eight primitive data types supported by the Java programming language are:

Type	Contains	Default	Size	Range
<code>boolean</code>	true or false	false	at least 1 bit	NA
<code>char</code>	Unicode character	<code>\u0000</code>	16 bits	<code>\u0000</code> to <code>\uFFFF</code>
<code>byte</code>	Signed integer	0	8 bits	-128 to 127
<code>short</code>	Signed integer	0	16 bits	-32768 to 32767
<code>int</code>	Signed integer	0	32 bits	-2147483648 to 2147483647
<code>long</code>	Signed integer	0L	64 bits	-9223372036854775808 to 9223372036854775807
<code>float</code>	IEEE 754 floating point	0.0F	32 bits (7 digits)	$\pm 1.4\text{E-}45$ to $\pm 3.4028235\text{E+}38$
<code>double</code>	IEEE 754 floating point	0.0	64 bits (15 digits)	$\pm 4.9\text{E-}324$ to $\pm 1.7976931348623157\text{E+}308$

**byte:** The `byte` data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The `byte` data type can be useful for saving memory in large [arrays](#), where the memory savings actually matters. They can also be used in place of `int` where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.

**short:** The `short` data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with `byte`, the same guidelines apply: you can use a `short` to save memory in large arrays, in situations where the memory savings actually matters.

**int:** By default, the `int` data type is a 32-bit signed two's complement integer, which has a minimum value of  $-2^{31}$  and a maximum value of  $2^{31}-1$ . In Java SE 8 and later, you can use the `int` data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of  $2^{32}-1$ . Use the `Integer` class to use `int` data type as an unsigned integer. See the section [The Number Classes](#) for more information. Static methods like `compareUnsigned`, `divideUnsigned` etc have been added to the `Integer` class to support the arithmetic operations for unsigned integers.

**long:** The `long` data type is a 64-bit two's complement integer. The signed long has a minimum value of  $-2^{63}$  and a maximum value of  $2^{63}-1$ . In Java SE 8 and later, you can use the `long` data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of  $2^{64}-1$ . Use this data type when you need a range of values wider than those provided by `int`. The `Long` class also contains methods like `compareUnsigned`, `divideUnsigned` etc to support arithmetic operations for unsigned long.

**float:** The `float` data type is a single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the [Floating-Point Types, Formats, and Values](#) section of the Java Language Specification. As with the recommendations for `byte` and `short`, use a `float` (instead of `double`) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency. For that, you will need to use the [java.math.BigDecimal](#) class instead. [Numbers and Strings](#) covers `BigDecimal` and other useful classes provided by the Java platform.

**double:** The `double` data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the [Floating-Point Types, Formats, and Values](#) section of the Java Language Specification. For decimal values, this data type is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.

**boolean:** The `boolean` data type has only two possible values: `true` and `false`. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.

**char:** The `char` data type is a single 16-bit Unicode character. It has a minimum value of `'\u0000'` (or 0) and a maximum value of `'\uffff'` (or 65,535 inclusive).

### String objects

In addition to the eight primitive data types listed above, the Java programming language also provides special support for character strings via the [java.lang.String](#) class. Enclosing your character string within double quotes will automatically create a new `String` object; for example, `String s = "this is a string";`. `String` objects are *immutable*, which means that once created, their values cannot be changed. The `String` class is not technically a primitive data type, but considering the special support given to it by the language, you'll probably tend to think of it as such.

## 7 Literals

A literal is the source code representation of a fixed value

### integer literals

An integer literal is of type `long` if it ends with the letter `L` or `l`; otherwise it is of type `int`. It is recommended that you use the upper case letter `L` because the lower case letter `l` is hard to distinguish from the digit `1`.

```
// The number 26, in decimal
int decVal = 26;
// The number 26, in hexadecimal
int hexVal = 0x1a;
// The number 26, in binary (Java SE7)
int binVal = 0b11010;
```

### floating-point literals

A floating-point literal is of type `float` if it ends with the letter `F` or `f`; otherwise its type is `double` and it can optionally end with the letter `D` or `d`.

```
double d1 = 123.4;
// same value as d1, but in scientific notation
double d2 = 1.234e2;
float f1 = 123.4f;
```

### Using Underscore Characters in Numeric Literals (Java SE 7)

In Java SE 7 and later, any number of underscore characters (`_`) can appear anywhere between digits in a numerical literal. This feature enables you, for example, to separate groups of digits in numeric literals, which can improve the readability of your code.

For instance, if your code contains numbers with many digits, you can use an underscore character to separate digits in groups of three, similar to how you would use a punctuation mark like a comma, or a space, as a separator.

The following example shows other ways you can use the underscore in numeric literals:

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

You can place underscores only between digits; you cannot place underscores in the following places:

- At the beginning or end of a number
- Adjacent to a decimal point in a floating point literal
- Prior to an `F` or `L` suffix
- In positions where a string of digits is expected

### Boolean Literals

There are only two possible boolean literals: `true` and `false`

### Character ('c') and String Literals ("string")

Literals of types `char` and `String` may contain any Unicode (UTF-16) characters. If your editor and file system allow it, you can use such characters directly in your code. If not, you can use a "Unicode escape" such as `'\u0108'` (capital C with circumflex), or `"S\u00ED Se\u00F1or"` (Sí Señor in Spanish). Always use 'single quotes' for `char` literals and



"double quotes" for `String` literals. Unicode escape sequences may be used elsewhere in a program (such as in field names, for example), not just in `char` or `String` literals.

The Java programming language also supports a few special **escape sequences**

for `char` and `String` literals: `\b` (backspace), `\t` (tab), `\n` (line feed), `\f` (form feed), `\r` (carriage return), `\"` (double quote), `\'` (single quote), and `\\` (backslash).

## Null literal

There's also a special **null** literal that can be used as a value for any reference type. **null** may be assigned to any variable, except variables of primitive types. There's little you can do with a **null** value beyond testing for its presence. Therefore, **null** is often used in programs as a marker to indicate that some object is unavailable.

## Class literals

Finally, there's also a special kind of literal called a *class literal*, formed by taking a type name and appending **.class** for example, **String.class**. This refers to an object (of type **java.lang.Class<String>**) that represents the type itself.

## 8 All variables are value typed

### Primitive Variables

- The primitive variable are value-typed. The value of a primitive variable is a literal (a fixed value).
- You cannot create a pointer or a reference to a primitive.
- Primitive variable assignment is by value
- Primitive parameter passing is by value

### Reference Variables

#### Objects (Class/Enum/Interface)

- An object is an instance of a concrete class (and not of abstract class or interface)
- A variable is not an object
- Variables that reference objects can be declared as a class, abstract class, enum, or interface.
- Object variables are value-typed
- The value of a reference variable is a *reference value* (aka a pointer or object identifier) to a specific object.
- Every object has a unique *reference value* (identifier)
- The Java reference value is not accessible to the developer.
- More than one variable can be set to the same object.
- Reference variable assignment is by value (ie the object's **reference (value)** or identifier)
- After the reference variable assignment **`x=y`**, `x` and `y` refer to the exact same specific object. Thus, changing `x` will result in changing `y`. Changing `y` will result in changing `x`.
- Reference parameter passing to methods is by value (ie the object reference)
- Neither **`Object.hashCode()`** nor **`System.identityHashCode`** give the object identifier
- You cannot create a memory address pointer to a reference variable, in contrast to C++.
- To perform object value assignment, you need to write a copy method that creates a new instance and then manually copies the state data. Copying can be shallow or deep.

#### Strings

- Strings are Objects
- String variables are reference variables.
- String variables are value-typed
- String objects are immutable.
- For every string literal, Java immediately creates an instance of String
- Java optimizes and if a string literal is encountered that has the exact same characters as an extant (aka live) String object, Java may use the object instead of creating a new object.

## Arrays

- Array variables are reference variables
- Arrays are objects but special objects. You create with a **new**
- Array variables are value-typed

## Type Variables

- A **type variable** is an unqualified identifier used as a type in a generic class, interface, method, and constructor bodies.

## 9 VARIABLES: BASIC DEFINITIONS

A *variable* is a storage location and has an associated type, sometimes called its *compile-time type*, that is either a *primitive type* (§4.2) or a *reference type* (§4.3).

A variable's value is changed by an assignment (§15.26) or by a prefix or postfix ++ (increment) or -- (decrement) operator (§15.14.2, §15.14.3, §15.15.1, §15.15.2).

Compatibility of the value of a variable with its type is guaranteed by the design of the Java programming language, as long as a program does not give rise to compile-time unchecked warnings (§4.12.2). Default values (§4.12.5) are compatible and all assignments to a variable are checked for assignment compatibility (§5.2), usually at compile time, but, in a single case involving arrays, a run-time check is made (§10.5). The type arguments of Generic types are not checked at run-time.

### 9.1 Variables of Primitive Type (4.12.1)

A variable of a primitive type always holds a primitive value (ie a literal) of that exact primitive type.

A primitive variable must be declared and initialized before it is used. This may be done in two separate steps

```
int x;    // declaration of x as a variable of primitive type int
x = 12;   // initialization of x
```

Or it can be performed in a single step.

```
int x = 12;
```

There is no default initialization of primitive types  
However, array elements do have a default initialization.

### 9.2 Variables of Reference Type (4.12.2)

A reference type variable must be declared and initialized before it is used. This may be done in two separate steps

```
String s;    // declaration of s as a variable of class type String
s = "Hi";    // initialization of s
```

Or it can be performed in a single step.

```
String s = "Hi";
```

#### EXAMPLE 2

```
public class Bicycle {
    public Bicycle() {}
}
```

This may be done in two separate steps

```
Bicycle b;    // declaration of s as a variable of class type String
b = new Bicycle();    // initialization of s
```

Or it can be performed in a single step.

```
Bicycle b = new Bicycle();
```

There is no default initialization of reference types  
However, array elements do have a default initialization.

NOTE: T below does not refer to a parameterized type, rather this is the Type of the variable declaration.

A **variable of a class type *T*** can hold a **null** reference or a reference to an instance of class *T* or of any class that is a subclass of *T*. The variable cannot hold a reference to an object of abstract class because such a class can not be instantiated.

```
String myvar;
```

A **variable of an interface type *T*** can hold a **null** reference or a reference to any instance of any class that implements the interface. The variable cannot hold a reference to the interface nor the class itself.

*Note that a variable is not guaranteed to always refer to a subtype of its declared type, but only to subclasses or subinterfaces of the declared type. This is due to the possibility of heap pollution discussed below.*

```
Cloneable myvar;
```

If ***T* is a primitive type**, then a variable of type "**array of *T***" can hold a **null** reference or a reference to any array of type "array of *T*".

```
int[] x;
```

If ***T* is a reference type**, then a variable of type "**array of *T***" can hold a **null** reference or a reference to any array of type "array of *S*" such that type *S* is a subclass or subinterface of type *T*.

```
String[] a;
```

A **variable of type *Object[]*** can hold a reference to an array of any reference type.

```
Object[] a;
```

A **variable of type *Object*** can hold a null reference or a reference to any object, whether it is an instance of a class or an array.

```
Object a;
```

# 10 Class Declaration, Object Instantiation, and Variable Declaration

## 10.1 Simple Classes and variables of class type

- `class Point { int[] metrics; }` is termed a **class (type) declaration**. This defines the class `Point`.
- `Point` is termed a **class (type)**
- `Point p;` is termed a **variable declaration**. This defines the variable `p`.  
`p` is termed a
  - variable of reference type
  - variable of class type
  - variable of class (type) `Point`
- `new Point()` is termed a class instance creation expression  
the **object** created by `new Point()` is called an **instance of class `Point`**
- `Point p = new Point();` is termed a *variable declaration with initializer*.
- Notes:
  - `p` is a *variable* of class type `Point`.
  - the value of `p` is a *reference* to an *instance* of class type `Point` (aka a `Point` object)
  - `p` itself is *not* an object!
  - A class that cannot be instantiated is called **abstract**
  - A class that can be instantiated is called **concrete**

## 10.2 Generic Classes

- `class Point<T> { T[] metrics; }` is termed a **generic class (type) declaration**.
- `Point<T>` is a **generic class type**
- `T` is termed a
  - **type variable**
  - **formal type parameter of the generic class `Point`**
- `Point<Integer>` is a **parameterized class type**<sup>1</sup>.
  - `T` has been replaced by an actual class, in this case `Integer`
  - In this case `Integer` is called the **type argument** for **parameterized class `Point`**
  - A **type argument** must be a reference type (**class**, **enum**, or **interface**). It *cannot* be a primitive.
- To **instantiate an object** of `Point`, the parameterized argument must be a **concrete class**:
  - `new Point<Integer>(11);`
- The parameterized argument *for the variable type declaration* (to the left of the variable) can be one of
  - `Point<Integer> p = new Point<Integer>(11);`  
the class with the same concrete type

---

<sup>1</sup> *generic type* and *parameterized type* are often used interchangeably

- `Point<Number> p = new Point<Integer>(11);`  
A superclass type (can be *abstract*, *concrete*) or an *interface* type
- `Point<Object> p = new Point<Integer>(11);`  
The `Object` class
- `Point<?> p = new Point<Integer>(11);`  
The wild card `?` symbol
- `Point p = new Point<Integer>(11);`  
No argument.  
Use of the class name without a parameterized argument is called a **raw type**.  
Use of raw types is **deprecated**, and required for backwards compatibility before generics were introduced.  
Use of a raw type will result in the following compiler warning  
**WARNING: Point is a raw type. References to generic type Point<T> should be parameterized**  
**Use wildcard instead of raw type.** Wildcard provides the identical functionality without a warning.

There are differences between `Point<Object> p` and `Point<?> p`

# 11 Kinds of Variables

The Java programming language defines the following kinds of variables:

- **Instance Variables (Non-Static Fields)** Technically speaking, objects store their individual states in "non-static fields", that is, fields declared without the `static` keyword. Non-static fields are also known as *instance variables* because their values are unique to each *instance* of a class (to each object, in other words); the `currentSpeed` of one bicycle is independent from the `currentSpeed` of another.
- **Class Variables (Static Fields)** A *class variable* is any field declared with the `static` modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated. A field defining the number of gears for a particular kind of bicycle could be marked as `static` since conceptually the same number of gears will apply to all instances. The code `static int numGears = 6;` would create such a static field. Additionally, the keyword `final` could be added to indicate that the number of gears will never change.
- **Local Variables** Similar to how an object stores its state in fields, a method will often store its temporary state in *local variables*. The syntax for declaring a local variable is similar to declaring a field (for example, `int count = 0;`). There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared — which is between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.
- **Parameters** You've already seen examples of parameters, both in the `Bicycle` class and in the `main` method of the "Hello World!" application. Recall that the signature for the `main` method is `public static void main(String[] args)`. Here, the `args` variable is the parameter to this method. The important thing to remember is that parameters are always classified as "variables" not "fields". This applies to other parameter-accepting constructs as well (such as constructors and exception handlers) that you'll learn about later in the tutorial.



# 12 Cast Expressions

## 15.16. Cast Expressions

A cast expression converts, *at run time*, a value of one numeric type to a similar value of another numeric type; or confirms, at compile time, that the type of an expression is **boolean**; or checks, at run time, that a reference value refers to an object either whose class is compatible with a specified reference type or list of reference types, or which embodies a value of a primitive type.

CastExpression:

```
( PrimitiveType ) UnaryExpression
( ReferenceType {& InterfaceType} ) UnaryExpressionNotPlusMinus
( ReferenceType {& InterfaceType} ) LambdaExpression
```

The parentheses and the type or list of types they contain are sometimes called the **cast operator**.

If the cast operator contains a list of types, that is, a **ReferenceType** followed by **one or more** **AdditionalBound** {& [InterfaceType](#)} terms, then all of the following must be true, or a **compile-time error** occurs:

- **ReferenceType** must denote a class or interface type.
- The erasures ([§4.6](#)) of all the listed types must be pairwise different.
- No two listed types may be subtypes of different parameterizations of the same generic interface.

The target type for the casting context ([§5.5](#)) introduced by the cast expression is either the *PrimitiveType* or the *ReferenceType* (if not followed by *AdditionalBound* terms) appearing in the cast operator, or the intersection type denoted by the *ReferenceType* and *AdditionalBound* terms appearing in the cast operator.

The type of a cast expression is the result of applying capture conversion ([§5.1.10](#)) to this target type.

Casts can be used to explicitly "tag" a lambda expression or a method reference expression with a particular target type. To provide an appropriate degree of flexibility, the target type may be a list of types denoting an intersection type, provided the intersection induces a functional interface ([§9.8](#)).

The result of a cast expression is not a variable, but a value, even if the result of evaluating the operand expression is a variable.

A cast operator has no effect on the choice of value set ([§4.2.3](#)) for a value of type **float** or type **double**. Consequently, a cast to type **float** within an expression that is not FP-strict ([§15.4](#)) does not necessarily cause its value to be converted to an element of the float value set, and a cast to type **double** within an expression that is not FP-strict does not necessarily cause its value to be converted to an element of the double value set.

If the compile-time type of the operand cannot be converted by casting conversion ([§5.5](#)) to the target type specified by the cast operator, then a compile-time error occurs.

Otherwise, at run time, the operand value is converted (if necessary) by casting conversion to the target type specified by the cast operator.

A `ClassCastException` is thrown if a cast is found at run time to be impermissible.

Some casts result in an error at compile time. Some casts can be proven, at compile time, always to be correct at run time. For example, it is always correct to convert a value of a class type to the type of its superclass; such a cast should require no special action at run time. Finally, some casts cannot be proven to be either always correct or always incorrect at compile time. Such casts require a test at run time. See [§5.5](#) for details.

## 13 Conversions and Contexts (including casting)

Every expression written in the Java programming language either produces no result (§15.1) or has a type that can be deduced at compile time (§15.3). When an expression appears in most contexts, it must be *compatible* with a type expected in that context; this type is called the *target type*. For convenience, compatibility of an expression with its surrounding context is facilitated in two ways:

- First, for some expressions, termed *poly expressions* (§15.2), the deduced type can be influenced by the target type. The same expression can have different types in different contexts.
- Second, after the type of the expression has been deduced, an implicit *conversion* from the type of the expression to the target type can sometimes be performed.

If neither strategy is able to produce the appropriate type, a compile-time error occurs.

The rules determining whether an expression is a poly expression, and if so, its type and compatibility in a particular context, vary depending on the kind of context and the form of the expression. In addition to influencing the type of the expression, the target type may in some cases influence the run time behavior of the expression in order to produce a value of the appropriate type.

Similarly, the rules determining whether a target type allows an implicit conversion vary depending on the kind of context, the type of the expression, and, in one special case, the value of a constant expression (§15.28). A conversion from type S to type T allows an expression of type S to be treated at compile time as if it had type T instead. In some cases this will require a corresponding action at run time to check the validity of the conversion or to translate the run-time value of the expression into a form appropriate for the new type T.

### Example 5.0-1. Conversions at Compile Time and Run Time

- A conversion from type *Object* to type *Thread* requires a run-time check to make sure that the run-time value is actually an instance of class *Thread* or one of its subclasses; if it is not, an exception is thrown.
- A conversion from type *Thread* to type *Object* requires no run-time action; *Thread* is a subclass of *Object*, so any reference produced by an expression of type *Thread* is a valid reference value of type *Object*.
- A conversion from type *int* to type *long* requires run-time sign-extension of a 32-bit integer value to the 64-bit *long* representation. No information is lost.
- A conversion from type *double* to type *long* requires a non-trivial translation from a 64-bit floating-point value to the 64-bit integer representation. Depending on the actual run-time value, information may be lost.

The conversions possible in the Java programming language are grouped into *several broad categories*:

- Identity conversions
- Widening primitive conversions
- Narrowing primitive conversions
- Widening reference conversions
- Narrowing reference conversions
- Boxing conversions
- Unboxing conversions
- Unchecked conversions
- Capture conversions
- String conversions
- Value set conversions

There are six kinds of **conversion contexts** in which poly expressions may be influenced by context or implicit conversions may occur. Each kind of context has different rules for poly expression typing and allows conversions in some of the categories above but not others. The contexts are:

- **Assignment contexts** (§5.2, §15.26), in which an expression's value is bound to a named variable. Primitive and reference types are subject to widening, values may be boxed or unboxed, and some primitive constant expressions may be subject to narrowing. An unchecked conversion may also occur.
- **Strict invocation contexts** (§5.3, §15.9, §15.12), in which an argument is bound to a formal parameter of a constructor or method. Widening primitive, widening reference, and unchecked conversions may occur.
- **Loose invocation contexts** (§5.3, §15.9, §15.12), in which, like strict invocation contexts, an argument is bound to a formal parameter. Method or constructor invocations may provide this context if no applicable declaration can be found using only strict invocation contexts. In addition to widening and unchecked conversions, this context allows boxing and unboxing conversions to occur.
- **String contexts** (§5.4, §15.18.1), in which a value of any type is converted to an object of type `String`.
- **Casting contexts** (§5.5), in which an expression's value is converted to a type explicitly specified by a cast operator (§15.16). Casting contexts are more inclusive than assignment or loose invocation contexts, allowing any specific conversion other than a string conversion, but certain casts to a reference type are checked for correctness at run time.
- **Numeric contexts** (§5.6), in which the operands of a numeric operator may be widened to a common type so that an operation can be performed.

The term "conversion" is also used to describe, without being specific, any conversions allowed in a particular context. For example, we say that an expression that is the initializer of a local variable is subject to "assignment conversion", meaning that a specific conversion will be implicitly chosen for that expression according to the rules for the assignment context.

**Example 5.0-2. Conversions In Various Contexts**

```
class Test {
    public static void main(String[] args) {
        // Casting conversion (5.4) of a float literal to
        // type int. Without the cast operator, this would
        // be a compile-time error, because this is a
        // narrowing conversion (5.1.3):
        int i = (int)12.5f;

        // String conversion (5.4) of i's int value:
        System.out.println("(int)12.5f==" + i);

        // Assignment conversion (5.2) of i's value to type
        // float. This is a widening conversion (5.1.2):
        float f = i;

        // String conversion of f's float value:
        System.out.println("after float widening: " + f);

        // Numeric promotion (5.6) of i's value to type
        // float. This is a binary numeric promotion.
        // After promotion, the operation is float*float:
        System.out.print(f);
        f = f * i;

        // Two string conversions of i and f:
        System.out.println("*" + i + "==" + f);

        // Invocation conversion (5.3) of f's value
        // to type double, needed because the method Math.sin
        // accepts only a double argument:
        double d = Math.sin(f);

        // Two string conversions of f and d:
        System.out.println("Math.sin(" + f + ")==" + d);
    }
}
```

*This program produces the output:*

```
(int)12.5f==12
after float widening: 12.0
12.0*12==144.0
Math.sin(144.0)=-0.49102159389846934
```

## 14 CASTING PRIMITIVES

### 14.1 widening casting

Widening is to convert a

19 specific conversions on primitive types are called the *widening primitive conversions*:

- **byte** to **short, int, long, float, or double**
- **short** to **int, long, float, or double**
- **char** to **int, long, float, or double**
- **int** to **long, float, or double**
- **long** to **float or double**
- **float** to **double**

The numeric value is preserved exactly for the above highlighted conversions. The other conversions are described below.

A widening primitive conversion from **float** to **double** in a **strictfp** expression (§15.4) preserves numeric value exactly.

A widening primitive conversion from **float** to **double** that is *not* **strictfp** may lose information about the overall magnitude of the converted value.

A widening primitive conversion from **int** to **float**, or from **long** to **float**, or from **long** to **double**, may result in *loss of precision* - that is, the result may lose some of the least significant bits of the value. In this case, the resulting floating-point value will be a correctly rounded version of the integer value, using IEEE 754 round-to-nearest mode (§4.2.4).

A widening conversion of a signed integer value to an integral type **T** simply sign-extends the two's-complement representation of the integer value to fill the wider format.

A widening conversion of a **char** to an integral type **T** zero-extends the representation of the **char** value to fill the wider format.

Despite the fact that loss of precision may occur, a widening primitive conversion never results in a run-time exception (§11.1.1).

### 14.2 narrowing casting

22 specific conversions on primitive types are called the *narrowing primitive conversions*:

- **short** to **byte or char**
- **char** to **byte or short**
- **int** to **byte, short, or char**
- **long** to **byte, short, char, or int**
- **float** to **byte, short, char, int, or long**
- **double** to **byte, short, char, int, long, or float**

A narrowing primitive conversion may lose information about the overall magnitude of a numeric value and may also lose precision and range.

A narrowing primitive conversion from **double** to **float** is governed by the IEEE 754 rounding rules (§4.2.4). This conversion can lose precision, but also lose range, resulting in a **float** zero from a nonzero **double** and a **float** infinity from a finite **double**. A **double** NaN is converted to a **float** NaN and a **double** infinity is converted to the same-signed **float** infinity.

A narrowing conversion of a signed **integer** to an integral type **T** simply discards all but the  $n$  lowest order bits, where  $n$  is the number of bits used to represent type **T**. In addition to a possible loss of information about the magnitude of the numeric value, this may cause the sign of the resulting value to differ from the sign of the input value.

A narrowing conversion of a **char** to an integral type **T** likewise simply discards all but the  $n$  lowest order bits, where  $n$  is the number of bits used to represent type **T**. In addition to a possible loss of information about the magnitude of the numeric value, this may cause the resulting value to be a negative number, even though **char**s represent 16-bit unsigned **integer** values.

A narrowing conversion of a floating-point number to an integral type **T** takes two steps:

**Step 1:** the floating-point number is converted either to a **long**, if **T** is **long**, or to an **int**, if **T** is **byte, short, char, or int**, as follows:

- If the floating-point number is NaN ([§4.2.3](#)), the result of the first step of the conversion is an **int** or **long** 0.
- Otherwise, if the floating-point number is not an infinity, the floating-point value is rounded to an **integer** value  $v$ , rounding toward zero using IEEE 754 round-toward-zero mode ([§4.2.3](#)). Then there are two cases:
  - If **T** is **long**, and this **integer** value can be represented as a **long**, then the result of the first step is the **long** value  $v$ .
  - Otherwise, if this **integer** value can be represented as an **int**, then the result of the first step is the **int** value  $v$ .
- Otherwise, one of the following two cases must be true:
  - The value must be too small (a negative value of large magnitude or negative infinity), and the result of the first step is the smallest representable value of type **int** or **long**.
  - The value must be too large (a positive value of large magnitude or positive infinity), and the result of the first step is the largest representable value of type **int** or **long**.

**Step 2**

- If **T** is **int** or **long**, the result of the conversion is the result of the first step.
- If **T** is **byte**, **char**, or **short**, the result of the conversion is the result of a narrowing conversion to type **T** ([§5.1.3](#)) of the result of the first step.

Despite the fact that overflow, underflow, or other loss of information may occur, a narrowing primitive conversion never results in a run-time exception ([§11.1.1](#)).

## 14.3 Widening and Narrowing Primitive Conversion

The following conversion combines both widening and narrowing primitive conversions:

- **byte** to **char**

First, the **byte** is converted to an **int** via widening primitive conversion ([§5.1.2](#)), and then the resulting **int** is converted to a **char** by narrowing primitive conversion ([§5.1.3](#)).

# 15 Wrapped primitives

Corresponding to each primitive there is an Object class that wraps the primitive in an object.

primitive	wrapper class
<b>boolean</b>	Boolean
<b>byte</b>	Byte
<b>short</b>	Short
<b>char</b>	Character
<b>int</b>	Integer
<b>long</b>	Long
<b>float</b>	Float
<b>double</b>	Double

- primitives must be compared using **==** operator

```
short x = 25;
short y = 34;
if (x == y) {
    ....
}
```

- wrapped primitives must be compared using **.equals** method

```
Short X = 25;
Short Y = 34;
if (X.equals(Y)) {
    ....
}
```

or (for example) using a value method

```
short x = 34;
Short X = 25;
if (x == X.shortValue()) {
    ....
}
```

```
Short X = 25;
Short Y = 34;
if (X.shortValue() == Y.shortValue()) {
    ....
}
```

## Valid initialization

```
✓ Short X = 34;  
✓ Short X = (short)34;  
✓ Short X = Short.valueOf((short)34);
```

## Deprecated initialization via constructor

```
✓ Short X = new Short((short)34);
```

## Compiler errors

```
✗ Short X = (Short) 25;  
✗ Short X = Short.valueOf(34);  
✗ Short X = new Short(34);
```



## 15.1 EXAMPLE

### Class Short

```
java.lang.Object
  java.lang.Number
    java.lang.Short
```

#### All Implemented Interfaces:

Serializable, Comparable<Short>

```
public final class Short
extends Number
implements Comparable<Short>
```

The Short class wraps a value of primitive type short in an object. An object of type Short contains a single field whose type is short.

#### 15.1.1 CLASS FIELDS

Fields	
Modifier and Type	Field and Description
static short	<b>MAX_VALUE</b> A constant holding the maximum value a short can have, $2^{15}-1$ .
static short	<b>MIN_VALUE</b> A constant holding the minimum value a short can have, $-2^{15}$ .
static int	<b>SIZE</b> The number of bits used to represent a short value in two's complement binary form.
static Class<Short>	<b>TYPE</b> The Class instance representing the primitive type short.

## 15.2 SELECTED CLASS METHODS

static String	<b>toString(short s)</b> Returns a new String object representing the specified short.
static Short	<b>valueOf(short s)</b> Returns a Short instance representing the specified short value.
static Short	<b>valueOf(String s)</b> Returns a Short object holding the value given by the specified String.

### ALL CONSTRUCTORS ARE DEPRECATED

## SELECTED INSTANCE METHODS

boolean	<b>equals(Object obj)</b> Compares this object to the specified object.
short	<b>shortValue()</b> Returns the value of this Short as a short.
String	<b>toString()</b> Returns a String object representing this Short's value.

#### 15.2.1

## 16 Primitive Boxing / Unboxing

**Boxing** refers to automatic run-time conversion from a primitive to a corresponding wrapper object.

- Special rule about object reference values created by boxing

If the value **p** being boxed is the result of evaluating a *constant expression* (§15.28) of type **boolean**, **char**, **short**, **int**, or **long**, and the result is **true**, **false**, a character in the range '**\u0000**' to '**\u007f**' inclusive, or an integer in the range –128 to 127 inclusive, then let **a** and **b** be the results of any two boxing conversions of **p**. It is always the case that **a == b**.

Ideally, boxing a primitive value would always yield an identical reference. In practice, this may not be feasible using existing implementation techniques. The rule above is a pragmatic compromise, *requiring that certain common values always be boxed into indistinguishable objects*. The implementation may cache these, lazily or eagerly. For other values, the rule disallows any assumptions about the identity of the boxed values on the programmer's part. This allows (but does not require) sharing of some or all of these references.

This ensures that in most common cases, the behavior will be the desired one, without imposing an undue performance penalty, especially on small devices. Less memory-limited implementations might, for example, cache all **char** and **short** values, as well as **int** and **long** values in the range of –32K to +32K.

**Unboxing** refers to automatic run-time conversion from a wrapper primitive to the corresponding primitive .

If **r** is **null**, unboxing conversion throws a `NullPointerException`

## 17 CASTING OBJECTS

Casting is performed by simply putting the class (or interface or enum) name in parentheses

```
castType variableName = (castType) variableToConvert;
```

# 18 Widening Casts of Objects

Widening is casting from an object from a class to a superclass.

explicit cast is automatic, but you can explicitly write it out

## Widening Reference Conversion (5.1.5)

A **widening reference conversion** exists from any reference type S to any reference type T, provided S is a subtype of T ([§4.10](#)).

Widening reference conversions never require a special action at run time and therefore never throw an exception at run time. They consist simply in regarding a reference as having some other type in a manner that can be proved correct at compile time.

*The null type is not a reference type ([§4.1](#)), and so a widening reference conversion does not exist from the null type to a reference type. However, many conversion contexts explicitly allow the null type to be converted to a reference type.*

Assume **class B extends A**

```
B b = new B();  
A a1 = (A) b;    // explicit widening cast  
A a2 = b;        // implicit widening cast
```

## 18.1 Widening as Object

Declaring a variable as Object effectively gives "duck typing". It can be assigned to any object. Via boxing any Object can be set to any primitive.

```
Object obj0 = new Object();  
obj0.getClass().getCanonicalName() --> "java.lang.Object"
```

```
Object obj1 = "Hello";  
obj1.getClass().getCanonicalName() --> "java.lang.String"
```

```
Object obj2 = 21;  
obj2.getClass().getCanonicalName() --> "java.lang.Integer"
```

## 19 Narrowing

Narrowing is casting from of an object from a class to a different class that is not a superclass, typically to a subclass.

This is typically performed as follows

```
S s = new S();  
if (S instanceof T) {  
    T t = (T) s;  
    ...  
}
```

## 20 Narrowing: Gory Details

A **narrowing reference conversion** treats expressions of a reference type **S** as expressions of a different reference type **T**, where **S** is *not* a subtype of **T**.

The supported pairs of types are defined in §5.1.6.1. Unlike widening reference conversion, the types need not be directly related. However, there are restrictions that prohibit conversion between certain pairs of types when it can be statically proven that no value can be of both types.

A narrowing reference conversion may require a test at run time to validate that a value of type **S** is a legitimate value of type **T**. However, due to the lack of parameterized type information at run time, some conversions cannot be fully validated by a run time test; they are flagged at compile time (§5.1.6.2). For conversions that can be fully validated by a run time test, and for certain conversions that involve parameterized type information but can still be partially validated at run time, a `ClassCastException` is thrown if the test fails (§5.1.6.3).

### 5.1.6.1. Allowed Narrowing Reference Conversion

A narrowing reference conversion exists from reference type **S** to reference type **T** if all of the three following conditions hold:

- **S** is not a subtype of **T** (§4.10)
- If there exists a parameterized type **X** that is a supertype of **T**, and a parameterized type **Y** that is a supertype of **S**, such that the erasures of **X** and **Y** are the same, then **X** and **Y** are not provably distinct (§4.5).

Using types from the `java.util` package as an example, no narrowing reference conversion exists from `ArrayList<String>` to `ArrayList<Object>`, or vice versa, because the type arguments `String` and `Object` are provably distinct. For the same reason, no narrowing reference conversion exists from `ArrayList<String>` to `List<Object>`, or vice versa. The rejection of provably distinct types is a simple static gate to prevent "stupid" narrowing reference conversions.

- One of the following cases applies:
  1. **S** and **T** are class types, and either  $|S| < : |T|$  or  $|T| < : |S|$ .
  2. **S** and **T** are interface types.
  3. **S** is a class type, **T** is an interface type, and **S** does not name a **final** class (§8.1.1).
  4. **S** is a class type, **T** is an interface type, and **S** names a **final** class that implements the interface named by **T**.
  5. **S** is an interface type, **T** is a class type, and **T** does not name a **final** class.
  6. **S** is an interface type, **T** is a class type, and **T** names a **final** class that implements the interface named by **S**.
  7. **S** is the class type `Object` or the interface type `java.io.Serializable` or `Cloneable` (the only interfaces implemented by arrays (§10.8)), and **T** is an array type.
  8. **S** is an array type `SC [ ]`, that is, an array of components of type **SC**; **T** is an array type `TC [ ]`, that is, an array of components of type **TC**; and a narrowing reference conversion exists from **SC** to **TC**.
  9. **S** is a type variable, and a narrowing reference conversion exists from the upper bound of **S** to **T**.
  10. **T** is a type variable, and either a widening reference conversion or a narrowing reference conversion exists from **S** to the upper bound of **T**.
  11. **S** is an intersection type  $S_1 \& \dots \& S_n$ , and for all  $i$  ( $1 \leq i \leq n$ ), either a widening reference conversion or a narrowing reference conversion exists from  $S_i$  to **T**.
  12. **T** is an intersection type  $T_1 \& \dots \& T_n$ , and for all  $i$  ( $1 \leq i \leq n$ ), either a widening reference conversion or a narrowing reference conversion exists from **S** to  $T_i$ .

### 5.1.6.2. Checked and Unchecked Narrowing Reference Conversions

A narrowing reference conversion is either **checked** or **unchecked**. These terms refer to the ability of the Java Virtual Machine to validate, or not, the type correctness of the conversion.

If a narrowing reference conversion is **unchecked**, then the Java Virtual Machine will not be able to fully validate its type correctness, possibly leading to heap pollution (§4.12.2). To flag this to the programmer, an unchecked narrowing reference conversion causes a compile-time *unchecked warning*, unless suppressed by `@SuppressWarnings` (§9.6.4.5).

In contrast, if a narrowing reference conversion is not unchecked, then it is **checked**; the Java Virtual Machine will be able to fully validate its type correctness, so no warning is given at compile time.

The **unchecked** narrowing reference conversions are as follows:

- A narrowing reference conversion from a type *S* to a parameterized class or interface type *T* is unchecked, unless at least one of the following is true:
  - All of the type arguments of *T* are unbounded wildcards.
  - $T \leq S$ , and *S* has no subtype *X* other than *T* where the type arguments of *X* are not contained in the type arguments of *T*.
- A narrowing reference conversion from a type *S* to a type variable *T* is unchecked.
- A narrowing reference conversion from a type *S* to an intersection type  $T_1 \& \dots \& T_n$  is unchecked if there exists a  $T_i$  ( $1 \leq i \leq n$ ) such that *S* is not a subtype of  $T_i$  and a narrowing reference conversion from *S* to  $T_i$  is unchecked.

### 5.1.6.3. Narrowing Reference Conversions at Run Time

All **checked** narrowing reference conversions require a validity check at run time. Primarily, these conversions are to class and interface types that are not parameterized.

Some **unchecked** narrowing reference conversions require a validity check at run time. This depends on whether the unchecked narrowing reference conversion is **completely unchecked** or **partially unchecked**.

A **partially unchecked** narrowing reference conversion requires a validity check at run time.

A **completely unchecked** narrowing reference conversion does not.

*These terms refer to the compatibility of the types involved in the conversion when viewed as raw types. If the conversion is conceptually an "upcast", then the conversion is completely unchecked; no run time test is needed because the conversion is legal in the non-generic type system of the Java Virtual Machine. In contrast, if the conversion is conceptually a "downcast", then the conversion is partially unchecked; even in the non-generic type system of the Java Virtual Machine, a run time check is needed to test the compatibility of the (raw) types involved in the conversion.*

*Using types from the `java.util` package as an example, a conversion from `ArrayList<String>` to `Collection<T>` is completely unchecked, because the (raw) type `ArrayList` is a subtype of the (raw) type `Collection` in the Java Virtual Machine. In contrast, a conversion from `Collection<T>` to `ArrayList<String>` is partially unchecked, because the (raw) type `Collection` is not a subtype of the (raw) type `ArrayList` in the Java Virtual Machine.*

The categorization of an unchecked narrowing reference conversion is as follows:

- An unchecked narrowing reference conversion from **S** to a non-intersection type **T** is **completely unchecked** if  $|S| < |T|$ . Otherwise, it is **partially unchecked**.
- An unchecked narrowing reference conversion from **S** to an intersection type  $T_1 \& \dots \& T_n$  is **completely unchecked** if, for all  $i$  ( $1 \leq i \leq n$ ), either  $S <: T_i$  or a narrowing reference conversion from **S** to **T<sub>i</sub>** is completely unchecked. Otherwise, it is **partially unchecked**.

The **run time validity check** for a checked or partially unchecked narrowing reference conversion is as follows:

- If the value at run time is **null**, then the conversion is allowed.
- Otherwise, let **R** be the class of the object referred to by the value, and let **T** be the erasure (§4.6) of the type being converted to. Then:
  - If **R** is an ordinary class (not an array class):
    - If **T** is a class type, then **R** must be either the same class as **T** (§4.3.4) or a subclass of **T**, or a `ClassCastException` is thrown.
      - If **T** is an interface type, then **R** must implement interface **T** (§8.1.5), or a `ClassCastException` is thrown.
      - If **T** is an array type, then a `ClassCastException` is thrown.
    - If **R** is an interface:
  - Note that **R** cannot be an interface when these rules are first applied for any given conversion, but **R** may be an interface if the rules are applied recursively because the run-time reference value may refer to an array whose element type is an interface type.
    - If **T** is a class type, then **T** must be `Object` (§4.3.2), or a `ClassCastException` is thrown.
    - If **T** is an interface type, then **R** must be either the same interface as **T** or a subinterface of **T**, or a `ClassCastException` is thrown.
    - If **T** is an array type, then a `ClassCastException` is thrown.
  - If **R** is a class representing an array type **RC**[], that is, an array of components of type **RC**:
    - If **T** is a class type, then **T** must be `Object` (§4.3.2), or a `ClassCastException` is thrown.
    - If **T** is an interface type, then **T** must be the type `java.io.Serializable` or `Cloneable` (the only interfaces implemented by arrays), or a `ClassCastException` is thrown.
    - If **T** is an array type **TC**[], that is, an array of components of type **TC**, then a `ClassCastException` is thrown unless either **TC** and **RC** are the same primitive type, or **TC** and **RC** are reference types and are allowed by a recursive application of these run-time rules.

If the conversion is to an intersection type  $T_1 \& \dots \& T_n$ , then for all  $i$  ( $1 \leq i \leq n$ ), any run-time check required for a conversion from **S** to **T<sub>i</sub>** is also required for the conversion to the intersection type.



## 21 Assignment Conversion Context Details

- **Assignment contexts** (§5.2, §15.26), in which an expression's value is bound to a named variable. Primitive and reference types are subject to widening, values may be boxed or unboxed, and some primitive constant expressions may be subject to narrowing. An unchecked conversion may also occur.

### 5.2. Assignment Contexts

**Assignment contexts** allow the value of an expression to be assigned (§15.26) to a variable; the type of the expression must be converted to the type of the variable.

Assignment contexts allow the use of one of the following:

- an **identity conversion** (§5.1.1)
- a **widening primitive conversion** (§5.1.2)
- a **widening reference conversion** (§5.1.5)
- a **widening reference conversion followed by an unboxing conversion**
- a **widening reference conversion followed by an unboxing conversion**, then **followed by a widening primitive conversion**
- a **boxing conversion** (§5.1.7)
- a **boxing conversion followed by a widening reference conversion**
- an **unboxing conversion** (§5.1.8)
- an **unboxing conversion followed by a widening primitive conversion**

If, after the conversions listed above have been applied, the resulting type is a **raw type** (§4.8), an **unchecked conversion** (§5.1.9) may then be applied. An unchecked conversion is going from a raw class/interface to a parameterized type. A warning is issued. Use `<?>` instead of raw type to avoid warning.

In addition, if the expression is a **constant expression** (§15.28) of type **byte**, **short**, **char**, or **int**:

- A **narrowing primitive conversion** may be used if the variable is of type **byte**, **short**, or **char**, and the value of the constant expression is representable in the type of the variable.

*The compile-time narrowing of constant expressions means that code such as:*

```
byte theAnswer = 42;
```

*is allowed. Without the narrowing, the fact that the integer literal 42 has type int would mean that a cast to byte would be required:*

```
byte theAnswer = (byte)42; // cast is permitted but not required
```

- A **narrowing primitive conversion followed by a boxing conversion** may be used if the variable is of type `Byte`, `Short`, or `Character`, and the value of the constant expression is representable in the type **byte**, **short**, or **char** respectively. (not int)

Finally, a value of the **null** type (the null reference is the only such value) may be assigned to any reference type, resulting in a null reference of that type.

It is a **compile-time error** if the **chain of conversions** contains two parameterized types that are not in the subtype relation (§4.10).

*An example of such an illegal chain would be:*

```
Integer, Comparable<Integer>, Comparable, Comparable<String>
```

*The first three elements of the chain are related by widening reference conversion, while the last entry is derived from its predecessor by unchecked conversion. However, this is not a valid assignment conversion, because the chain contains two parameterized types, `Comparable<Integer>` and `Comparable<String>`, that are not subtypes.*

If the type of an expression can be converted to the type of a variable by assignment conversion, we say the expression (or its value) is **assignable** to the variable **or**, equivalently, that the type of the expression is **assignment compatible with** the type of the variable.

If the type of the variable is **float** or **double**, then **value set conversion** (§5.1.13) is applied to the value **v** that is the result of the conversion(s). *Value set conversion* is the process of mapping a floating-point value from one value set to another without changing its type, ie changing being FP-strict and not FP-strict.

- If **v** is of type **float** and is an element of the float-extended-exponent value set, then the implementation must map **v** to the nearest element of the float value set. This conversion may result in overflow or underflow.
- If **v** is of type **double** and is an element of the double-extended-exponent value set, then the implementation must map **v** to the nearest element of the double value set. This conversion may result in overflow or underflow.

The only **exceptions** that may arise from conversions in an assignment context are:

- A **ClassCastException** if, after the conversions above have been applied, the resulting value is an object which is not an instance of a subclass or subinterface of the erasure (§4.6) of the type of the variable.  
*This circumstance can only arise as a result of heap pollution (§4.12.2). In practice, implementations need only perform casts when accessing a field or method of an object of parameterized type when the erased type of the field, or the erased return type of the method, differ from its unerased type.*
- An **OutOfMemoryError** as a result of a boxing conversion.
- A **NullPointerException** as a result of an unboxing conversion on a null reference.
- An **ArrayStoreException** in special cases involving array elements or field access (§10.5, §15.26.1).

**Example 5.2-1. Assignment for Primitive Types**

```
class Test {
    public static void main(String[] args) {
        short s = 12;        // narrow 12 to short
        float f = s;         // widen short to float
        System.out.println("f=" + f);
        char c = '\u0123';
        long l = c;          // widen char to long
        System.out.println("l=0x" + Long.toString(l,16));
        f = 1.23f;
        double d = f;        // widen float to double
        System.out.println("d=" + d);
    }
}
```

*This program produces the output:*

```
f=12.0
l=0x123
d=1.2300000190734863
```

*The following program, however, produces compile-time errors:*

```
class Test {
    public static void main(String[] args) {
        short s = 123;
        char c = s;    // error: would require cast
        s = c;         // error: would require cast
    }
}
```

*because not all short values are char values, and neither are all char values short values.*

**Example 5.2-2. Assignment for Reference Types**

```
class Point { int x, y; }
class Point3D extends Point { int z; }
interface Colorable { void setColor(int color); }

class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}

class Test {
    public static void main(String[] args) {
        // Assignments to variables of class type:
        Point p = new Point();
        p = new Point3D();
        // OK because Point3D is a subclass of Point
        Point3D p3d = p;
        // Error: will require a cast because a Point
        // might not be a Point3D (even though it is,
        // dynamically, in this example.)

        // Assignments to variables of type Object:
        Object o = p;          // OK: any object to Object
        int[] a = new int[3];
        Object o2 = a;         // OK: an array to Object

        // Assignments to variables of interface type:
        ColoredPoint cp = new ColoredPoint();
        Colorable c = cp;
        // OK: ColoredPoint implements Colorable

        // Assignments to variables of array type:
        byte[] b = new byte[4];
        a = b;
        // Error: these are not arrays of the same primitive type
        Point3D[] p3da = new Point3D[3];
        Point[] pa = p3da;
        // OK: since we can assign a Point3D to a Point
        p3da = pa;
        // Error: (cast needed) since a Point
        // can't be assigned to a Point3D
    }
}
```

The following test program illustrates assignment conversions on reference values, but fails to compile, as described in its comments. This example should be compared to the preceding one.

```
class Point { int x, y; }
interface Colorable { void setColor(int color); }
class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}

class Test {
    public static void main(String[] args) {
        Point p = new Point();
        ColoredPoint cp = new ColoredPoint();
        // Okay because ColoredPoint is a subclass of Point:
        p = cp;
        // Okay because ColoredPoint implements Colorable:
        Colorable c = cp;
        // The following cause compile-time errors because
        // we cannot be sure they will succeed, depending on
        // the run-time type of p; a run-time check will be
        // necessary for the needed narrowing conversion and
        // must be indicated by including a cast:
        cp = p;    // p might be neither a ColoredPoint
                  // nor a subclass of ColoredPoint
        c = p;    // p might not implement Colorable
    }
}
```

**Example 5.2-3. Assignment for Array Types**

```

class Point { int x, y; }
class ColoredPoint extends Point { int color; }

class Test {
    public static void main(String[] args) {
        long[] vecLong = new long[100];
        Object o = vecLong;           // okay
        Long l = vecLong;             // compile-time error
        short[] vecShort = vecLong;   // compile-time error
        Point[] pvec = new Point[100];
        ColoredPoint[] cpvec = new ColoredPoint[100];
        pvec = cpvec;                 // okay
        pvec[0] = new Point();         // okay at compile time,
                                     // but would throw an
                                     // exception at run time
        cpvec = pvec;                 // compile-time error
    }
}

```

In this example:

- The value of `vecLong` cannot be assigned to a `Long` variable, because `Long` is a class type other than `Object`. An array can be assigned only to a variable of a compatible array type, or to a variable of type `Object`, `Cloneable` or `java.io.Serializable`.
- The value of `vecLong` cannot be assigned to `vecShort`, because they are arrays of primitive type, and `short` and `Long` are not the same primitive type.
- The value of `cpvec` can be assigned to `pvec`, because any reference that could be the value of an expression of type `ColoredPoint` can be the value of a variable of type `Point`. The subsequent assignment of the new `Point` to a component of `pvec` then would throw an `ArrayStoreException` (if the program were otherwise corrected so that it could be compiled), because a `ColoredPoint` array cannot have an instance of `Point` as the value of a component.
- The value of `pvec` cannot be assigned to `cpvec`, because not every reference that could be the value of an expression of type `Point` can correctly be the value of a variable of type `ColoredPoint`. If the value of `pvec` at run time were a reference to an instance of `Point[]`, and the assignment to `cpvec` were allowed, a simple reference to a component of `cpvec`, say, `cpvec[0]`, could return a `Point`, and a `Point` is not a `ColoredPoint`. Thus to allow such an assignment would allow a violation of the type system. A cast may be used (§5.5, §15.16) to ensure that `pvec` references a `ColoredPoint[]`:

```

cpvec = (ColoredPoint[])pvec; // OK, but may throw an
                             // exception at run time

```

## 22 Casting Conversion Context Details

**Casting contexts** (§5.5), in which an expression's value is converted to a type explicitly specified by a cast operator (§15.16).

Casting contexts are more inclusive than assignment or loose invocation contexts, allowing any specific conversion other than a string conversion, but certain casts to a reference type are checked for correctness at run time.

### 5.5. Casting Contexts

Casting contexts allow the operand of a cast expression (§15.16) to be converted to the type explicitly named by the cast operator.

Compared to assignment contexts and invocation contexts, casting contexts allow the use of more of the conversions defined in §5.1, and allow more combinations of those conversions.

If the expression is of a **primitive type**, then a casting context allows the use of one of the following:

- an identity conversion (§5.1.1)
- a widening primitive conversion (§5.1.2)
- a narrowing primitive conversion (§5.1.3)
- a widening and narrowing primitive conversion (§5.1.4)
- a boxing conversion (§5.1.7)
- a boxing conversion followed by a widening reference conversion (§5.1.5)

If the expression is of a **reference type**, then a casting context allows the use of one of the following:

- an identity conversion (§5.1.1)
- a widening reference conversion (§5.1.5)
- a widening reference conversion followed by an unboxing conversion
- a widening reference conversion followed by an unboxing conversion, then followed by a widening primitive conversion
- a narrowing reference conversion (§5.1.6)
- a narrowing reference conversion followed by an unboxing conversion
- an unboxing conversion (§5.1.8)
- an unboxing conversion followed by a widening primitive conversion

If the expression has the **null** type, then the expression may be cast to any reference type.

If a casting context makes use of a narrowing reference conversion that is checked or partially unchecked (§5.1.6.2, §5.1.6.3), then a run time check will be performed on the class of the expression's value, possibly causing a `ClassCastException`. Otherwise, no run time check is performed.

Value set conversion (§5.1.13) is applied after the type conversion.

## 23 NumericConversion Context Details

- **Numeric contexts** (§5.6), in which the operands of a numeric operator may be widened to a common type so that an operation can be performed.
- 

### 5.6. Numeric Contexts

Numeric contexts apply to the operands of an **arithmetic operator**.

Numeric contexts allow the use of:

- an **identity conversion** (§5.1.1)
- a **widening primitive conversion** (§5.1.2)
- a **widening reference conversion** (§5.1.5) followed by an **unboxing conversion**
- a **widening reference conversion** followed by an **unboxing conversion**, then followed by a **widening primitive conversion**
- an **unboxing conversion** (§5.1.8)
- an **unboxing conversion** followed by a **widening primitive conversion**

A **numeric promotion** is a process by which, given an arithmetic operator and its argument expressions, the arguments are converted to an inferred target type **T**. **T** is chosen during promotion such that each argument expression can be converted to **T** and the arithmetic operation is defined for values of type **T**.

- The two kinds of numeric promotion are **unary numeric promotion** (§5.6.1) and **binary numeric promotion** (§5.6.2).

#### 5.6.1. Unary Numeric Promotion

Some operators apply *unary numeric promotion* to a single operand, which must produce a value of a numeric type:

- If the operand is of compile-time type **Byte**, **Short**, **Character**, or **Integer**, it is subjected to unboxing conversion (§5.1.8). The result is then *promoted to a value of type **int*** by a widening primitive conversion (§5.1.2) or an identity conversion (§5.1.1).
- Otherwise, if the operand is of compile-time type **Long**, **Float**, or **Double**, it is subjected to unboxing conversion (§5.1.8).
- Otherwise, if the operand is of compile-time type **byte**, **short**, or **char**, it is *promoted to a value of type **int*** by a widening primitive conversion (§5.1.2).
- Otherwise, a unary numeric operand remains as is and is not converted.

After the conversion(s), if any, value set conversion (§5.1.13) is then applied.

Unary numeric promotion is performed on expressions in the following situations:

- Each dimension expression in an array creation expression (§15.10.1)
- The index expression in an array access expression (§15.10.3)
- The operand of a unary plus operator **+** (§15.15.3)
- The operand of a unary minus operator **-** (§15.15.4)
- The operand of a bitwise complement operator **~** (§15.15.5)
- Each operand, separately, of a shift operator **<<**, **>>**, or **>>>** (§15.19).

A **long** shift distance (right operand) does not promote the value being shifted (left operand) to **long**.



## 5.6.2. Binary Numeric Promotion

When an operator applies *binary numeric promotion* to a pair of operands, each of which must denote a value that is convertible to a numeric type, the following rules apply, in order:

### IMPORTANT

1. If any operand is of a **reference type**, it is subjected to **unboxing** conversion ([§5.1.8](#)).
2. **Widening primitive conversion** ([§5.1.2](#)) is applied to convert either or both operands as specified by the following rules:
  - If either operand is of type **double**, the other is converted to **double**.
  - Otherwise, if either operand is of type **float**, the other is converted to **float**.Otherwise, if either operand is of type **long**, the other is converted to **long**.  
**Otherwise**, both operands are converted to type **int**.

After the conversion(s), if any, value set conversion ([§5.1.13](#)) is then applied to each operand.

Binary numeric promotion is performed on the operands of certain operators:

- The multiplicative operators **\***, **/**, and **%** ([§15.17](#))
- The addition and subtraction operators for numeric types **+** and **-** ([§15.18.2](#))
- The numerical comparison operators **<**, **<=**, **>**, and **>=** ([§15.20.1](#))
- The numerical equality operators **==** and **!=** ([§15.21.1](#))
- The integer bitwise operators **&**, **^**, and **|** ([§15.22.1](#))
- In certain cases, the conditional operator **? :** ([§15.25](#))

## 24 Instanceof operator

True if x can be cast to Y at runtime (without raising a ClassCastException)

Always false for x = null

x instanceof Y

- x **must be a reference variable** (or expression) **or null**
- x **cannot be a literal** (with the **exception of String literals and null**)
- x can be an expression as long as the result of the expression is a ReferenceType (not a PrimitiveType)
- Y must be the name of a class (or other ReferenceType) that is also **reifiable**
- **Compile error if**
  - **X** x is a primitive
  - **X** x is a literal (with the exception of string literals and null)
  - **X** Y is a primitive type
  - **X** x cannot be cast to Y at compile time
  - **X** Y is not reifiable

### Primitives

2 instanceof int → **X compile time error** (int is a primitive type)  
 (2.5) instanceof Object → **X compile time error** (same for literal or primitive variable)

### NULL

null instanceof Object → **false**

### Objects

String instanceof Object → **X compile time error** (String is a class not an object)  
 "hello" instanceof String → **true** (same for String literal or String variable)  
 "hello" instanceof Number → **false**  
 ("hello" + "!") instanceof Object → **true** (expressions are okay)  
 ("hello") instanceof String[] → **X compile time error** (hello is not an array)

```
Integer n = Integer.valueOf(23);
n instanceof Object → true
n instanceof Number → true
n instanceof Integer → true
n instanceof Long → false
```

```

Number n = Integer.valueOf(23);
n instanceof Object → true
n instanceof Number → true
n instanceof Integer → true
n instanceof Long → false

```

### Arrays of Primitives

```

new int[0] instanceof Object → true
new int[0] instanceof int[] → true (primitive arrays are ok)
new int[0] instanceof long[] → ✗ compile time error
new int[0] instanceof float[] → ✗ compile time error
new int[0] instanceof Integer[] → ✗ compile time error
new int[0] instanceof Object[] → ✗ compile time error

```

### Arrays of Objects

```

new String[0] instanceof Object → true
new String[0] instanceof Object[] → true
new String[0] instanceof String[] → true
new String[0] instanceof Integer[] → ✗ compile time error
new String[0] instanceof String → ✗ compile time error
new Integer[0] instanceof Number[] → true

```

### Generic Classes

```

new ArrayList<String> instanceof Object → true
new ArrayList<String> instanceof Object[] → ✗ compile time error
new ArrayList<String> instanceof ArrayList → true
new ArrayList<String> instanceof AbstractList → true (superclass)
new ArrayList<String> instanceof List → true (interface)
new ArrayList<String> instanceof Object → true
new ArrayList<String> instanceof ArrayList<?> → true
new ArrayList<String> instanceof ArrayList<String> → ✗ compile time error
new ArrayList<String> instanceof Set<?> → false

```

## 24.1 `null` is a reference value, the unique reference to nothing.

If you try to access the fields or methods of an object variable that is set to `null`, you will get a runtime error (or in some cases a compile error).

Thus it's important to keep track of any Object variables that are not initialized and make sure to check if not null before using:

```
if (object != null) {  
    object.method();  
}
```

Attempting to reference members of an object variable that is set to `null` is the number one causes of Java runtime errors!

## 24.2 Specification

### 15.20.2. Type Comparison Operator `instanceof`

`RelationalExpression instanceof ReferenceType`

The type of the `RelationalExpression` operand of the `instanceof` operator must be a

- reference type or
- `null` type

otherwise, a compile-time **error** occurs.

It is a compile-time error if the **ReferenceType** mentioned after the `instanceof` operator does not denote a reference type that is **reifiable** (§4.7).

If a cast of the **RelationalExpression** to the **ReferenceType** would be rejected as a compile-time error, then the `instanceof` relational expression likewise produces a compile-time error. In such a situation, the result of the `instanceof` expression could never be true.

At run time, the result of the `instanceof` operator is `true` if the value of the *RelationalExpression* is not `null` and the reference could be cast (§15.16) to the *ReferenceType* without raising a `ClassCastException`. Otherwise the result is `false`.

<https://docs.oracle.com/javase/specs/jls/se11/html/jls-4.html#jls-4.5.1>

## 25 Tokens, Expressions, Statement, Declarations, Blocks

**Tokens** are the identifiers (§3.8), keywords (§3.9), literals (§3.10), separators (§3.11), and operators (§3.12) of the Java programming language. That is, it is the input source, with whitespace removed, grouped into identifiers, keywords, separators, and operators.

**Statements** are roughly equivalent to sentences in natural languages. A **statement** forms a complete unit of execution. The following types of expressions can be made into a statement by terminating the expression with a semicolon (;).

- declarations statements
- **break**, **continue**, **return**
- statement expressions
- **if then**
- **while**
- **do**
- **switch**
- **try**
- **assert**
- block {}
- **synchronized**
- **throw**

Note that declaration statements are not included in the above list!

An **expression** is a grouping of tokens (ie variables, operators, and method invocations) *that evaluates to a primitive value or an object*.

```
1+2
3.9*x+3
new Object()
"Hello" + "There"
(Double)(12)
new int[] {i++, i*2+45, 0}
obj.toString()
```

An expression cannot be used as a statement unless it is a **StatementExpressions**

```
x+2; ❌
x++; ✅
```

- You can put parentheses ( ) around any **Expression** except a **StatementExpression**

**StatementExpression** are expressions that can also be used as statements

<a href="#">Assignment</a>	x=4
<a href="#">PreIncrementExpression</a>	++x
<a href="#">PreDecrementExpression</a>	--x
<a href="#">PostIncrementExpression</a>	x++
<a href="#">PostDecrementExpression</a>	x--
<a href="#">MethodInvocation</a>	obj.getName()
<a href="#">ClassInstanceCreationExpression</a>	new Object()

A **block statement** is a sequence of zero or more statements enclosed by curly brackets

```
{
    int x = 12;
    System.out.println(x);
}
```

Any place that a statement is allowed, it can be replaced with a block statement, but not vice versa.

**Declarations Statements** are outlier statements. They are an "executable sentence" like a statement but they have different rules than all other statements.

Any place that a declaration statement is allowed, it can be replaced with a statement, but not vice versa.

**What can you put in a block (or a method body)?**

#### Blocks

A *block* is a sequence of statements, local class declarations, and local variable declaration statements within braces.

*Block:*

```
{ [BlockStatements] }
```

*BlockStatements:*

```
BlockStatement {BlockStatement}
```

*BlockStatement:*

```
LocalVariableDeclarationStatement
ClassDeclaration
Statement
```

A block is executed by executing each of the local variable declaration statements and other statements in order from first to last (left to right). If all of these block statements complete normally, then the block completes normally. If any of these block statements complete abruptly for any reason, then the block completes abruptly for the same reason.

## 26 Labels

You can label any [Statement](#)

You cannot label a [LocalVariableDeclarationStatement](#)

You cannot label a [ClassDeclaration](#)

```
int x;
label1: x = 2;

label2: int y = 3; ❌ // declarations are not allowed to be labelled

label3: if (x==0) x++;

label4:
    if (x==0) {
        x++;
    } else {
        x--;
    }

label5: {
    int y = 0;
    y += x;
}
```

## 27 If statements

### NO THEN IS USED

Valid Examples

```
int y = 25;
Boolean flag = null;

if (y > 0)
    flag = true;

if (y > 0)
    flag = true;
else
    flag = false;

if (y > 0) {
    flag = true;
} else {
    flag = false;
}
```

### No elseif keyword

The "short if" rules (see next section) allow else if structures without needing an elseif keyword.

```
if (y > 0) {
    flag = false;
} else if (y < -100) {
    flag = true;
} else if (y == -5) {
    flag = true;
} else {
    flag = false;
}
```



## 28 Short Ifs

A "short if" is an if statement without an else.

"Short ifs" are not allowed in certain situations to eliminate ambiguity.

The following is valid Java. There are no "short ifs" and no ambiguity.

```
boolean flag = false;
if (x > 0)
    if (x > 10)
        flag = true;
    else
        flag = false;
else
    flag = true;
```

The following is also valid java code, but without the "short if" rules there is ambiguity as to which if the else belongs.

```
if (x > 0) if (x < 10) flag = true; else flag = false;
```

The following Java language rules

```
IfThenStatement:
    if ( Expression ) Statement
IfThenElseStatement:
    if ( Expression ) StatementNoShortIf else Statement
```

imply that the meaning of the above code is

```
if (x > 0)
    if (x < 10)
        flag = true;
    else
        flag = false;
```

That is, the else belongs to the inner if statement.

Let's test in Java to be sure

```
static Boolean shorty (int x) {
    Boolean flag = null;
    if (x > 0) if (x < 10) flag = true; else flag = false;
    return flag;
}

public static void main(String[] args) {
    System.out.println(shorty(-1));
    System.out.println(shorty(5));
    System.out.println(shorty(20));
}
```

The output is

```
null
true
false
```

## 29 Switch statement

A switch statement is in essence an if-then-else

- Equality is defined in terms of the `==` operator (except for `String`, then `equals` method is used)
- The case expressions must be constant expressions, ie consisting of literals and/or `final` variables.
- The switch variable must be a primitive, a boxed primitive, `enum` or a `String`.
- The switch variable *cannot* be equal to `null`.
- No code is allowed between the switch statement and the first case!

The type of the *Expression* must be `char`, `byte`, `short`, `int`, `Character`, `Byte`, `Short`, `Integer`, `String`, or an `enum` type (§8.9), or a compile-time error occurs.

The body of a `switch` statement is known as a *switch block*. Any statement immediately contained by the switch block may be labeled with one or more *switch labels*, which are `case` or `default` labels. Every `case` label has a `case` constant, which is either a constant expression or the name of an `enum` constant. Switch labels and their `case` constants are said to be *associated* with the `switch` statement.

Given a `switch` statement, all of the following must be true or a compile-time error occurs:

- Every `case` constant associated with the `switch` statement must be assignment compatible with the type of the `switch` statement's *Expression* (§5.2).
- The expression must be a constant expression
- If the type of the `switch` statement's *Expression* is an `enum` type, then every `case` constant associated with the `switch` statement must be an `enum` constant of that type.
- No two of the `case` constants associated with the `switch` statement have the same value.
- No `case` constant associated with the `switch` statement is `null`.
- At most one `default` label is associated with the `switch` statement.

A `switch` statement is executed by first evaluating the *Expression*.

If the *Expression* evaluates to `null`, a `NullPointerException` is thrown and the entire `switch` statement completes abruptly for that reason.

Otherwise, if the result is of type `Character`, `Byte`, `Short`, or `Integer`, it is subject to unboxing conversion (§5.1.8).

If evaluation of the *Expression* or the subsequent unboxing conversion (if any) completes abruptly for some reason, the `switch` statement completes abruptly for the same reason.

Otherwise, execution continues by comparing the value of the *Expression* with each `case` constant, and there is a choice:

If one of the `case` constants is equal to the value of the expression, then we say that the `case` label *matches*. Equality is defined in terms of the `==` operator (§15.21) unless the value of the expression is a `String`, in which case equality is defined in terms of the `equals` method of class `String`.

All statements after the matching `case` label in the `switch` block, if any, are executed in sequence.

If all these statements complete normally, or if there are no statements after the matching `case` label, then the entire `switch` statement completes normally.

If no `case` label matches but there is a `default` label, then all statements after the `default` label in the `switch` block, if any, are executed in sequence.

If all these statements complete normally, or if there are no statements after the `default` label, then the entire `switch` statement completes normally.

If no `case` label matches and there is no `default` label, then no further action is taken and the `switch` statement completes normally.

If any statement immediately contained by the *Block* body of the `switch` statement completes abruptly, it is handled as follows:

If execution of the *Statement* completes abruptly because of a `break` with no label, no further action is taken and the `switch` statement completes normally.

If execution of the *Statement* completes abruptly for any other reason, the `switch` statement completes abruptly for the same reason.

## Example

```
static void MySwitch1(Integer x) {  
    out.println("MySwitch1( x = "+x+" )");  
    final int y = 5;  
  
    switch (x) {  
        // NO CODE ALLOWED HERE  
        case 0:  
            out.println("    case 0");  
            break;  
        case (y*2+1):  
            out.println("    case (y*2+1)");  
            break;  
        default:  
            out.println("    default");  
            break;  
    }  
}
```

```
public static void main(String[] args) {  
    MySwitch1(0);  
    MySwitch1(11);  
    MySwitch1(12);  
}
```

```
MySwitch1( x = 0 )  
    case 0  
MySwitch1( x = 11 )  
    case (y*2+1)  
MySwitch1( x = 12 )  
    default
```

## Example

Once a case is matched, ALL code is executed until a **break** is encountered. Without breaks, all code will be executed.

```
static void MySwitch2(Integer x) {  
    out.println("MySwitch2( x = "+x+" )");  
    final int y = 5;  
    switch (x) {  
        case 0:  
            out.println("    case 0");  
        case (y*2+1):  
            out.println("    case (y*2+1)");  
        default:  
            out.println("    default");  
    }  
}
```

```
public static void main(String[] args) {  
    MySwitch2(0);  
    MySwitch2(11);  
    MySwitch2(12);  
}
```

```
MySwitch2( x = 0 )  
    case 0  
    case (y*2+1)  
    default  
MySwitch2( x = 11 )  
    case (y*2+1)  
    default  
MySwitch2( x = 12 )  
    default
```

## 30 Conditional (ie Boolean) Expressions

- Conditional expressions must be **boolean** or `Boolean`
- **boolean**/`Boolean` cannot be cast

```
int x = 1;  
boolean b = (boolean) x; ❌ compile error
```

- can accomplish as follows

```
int x = 1;  
boolean b = (x != 0); ❌ compile error
```

- Java boolean expressions utilize short-circuiting like C/C++. Expressions are evaluated left-to-right.
  - For a sequence of ORs (`||`) once a **true** is encountered, the value is determined and the rest of the expressions are not evaluated
  - For a sequence of ANDs (`&&`) once a **false** is encountered, the value is determined and the rest of the expressions are not evaluated

## 31 The ternary expression

```
BooleanExpression ? ExpressionTrue : ExpressionFalse
```

- The ternary expression cannot be used as a statement.
- `ExpressionTrue` and `ExpressionFalse` must evaluate to the same type
- The ternary expression returns a value of the same type of `ExpressionTrue` / `ExpressionFalse`
- `ExpressionFalse` can be another ternary, but `ExpressionTrue` cannot be a ternary
- The ternary uses short-circuiting:
  - If `BooleanExpression` evaluates to **true**
    - evaluate and return `ExpressionTrue`
    - `ExpressionFalse` is not evaluated.
  - If `BooleanExpression` evaluates to **false**
    - evaluate and return `ExpressionFalse`
    - `ExpressionTrue` is not evaluated.

```
int x;
String s1 = "true";
String s2 = "false";
String s;

x = 5;
s = (x==5) ? s1 : s2 ;
out.println("x = " + x );
out.println("  ( (x==5) ? s1 : s2 ) = " + s );

x = 1;
s = (x==5) ? s1 : s2 ;
out.println("x = " + x );
out.println("  ( (x==5) ? s1 : s2 ) = " + s );
```

```
x = 5
  ( (x==5) ? s1 : s2 ) = true
x = 1
  ( (x==5) ? s1 : s2 ) = false
```

This is in a general sense (not literally) what the ternary is

```
Type ternary( boolean b, Type expTrue, Type expFalse) {
    if (b) {
        return expTrue;
    } else {
        return expFalse;
    }
}
```

The conditional operator `?` : uses the boolean value of one expression to decide which of two other expressions should be evaluated.

*ConditionalExpression:*

[\*ConditionalOrExpression\*](#)

[\*ConditionalOrExpression\*](#) `?` *Expression* : [\*ConditionalExpression\*](#)

[\*ConditionalOrExpression\*](#) `?` *Expression* : [\*LambdaExpression\*](#)

The conditional operator is syntactically right-associative (it groups right-to-left).:

`a?b:c?d:e?f:g` means the same as `a?b:(c?d:(e?f:g))`.

The conditional operator has three operand expressions. `?` appears between the first and second expressions, and `:` appears between the second and third expressions.

- The first expression must be of type **boolean** or **Boolean**, or a compile-time error occurs.
- It is a compile-time error for either the second or the third operand expression to be an invocation of a **void** method.

*In fact, by the grammar of expression statements (§14.8), it is not permitted for a conditional expression to appear in any context where an invocation of a **void** method could appear.*

There are *three kinds* of conditional expressions, classified according to the second and third operand expressions:

- *boolean conditional expressions*
- *numeric conditional expressions*
- *reference conditional expressions.*

The classification rules are as follows:

- If both the second and the third operand expressions are *boolean expressions*, the conditional expression is a boolean conditional expression.  
For the purpose of classifying a conditional, the following expressions are boolean expressions:
  - An expression of a standalone form (§15.2) that has type **boolean** or **Boolean**.
  - A parenthesized **boolean** expression (§15.8.5).
  - A class instance creation expression (§15.9) for class **Boolean**.
  - A method invocation expression (§15.12) for which the chosen most specific method (§15.12.2.5) has return type **boolean** or **Boolean**.  
*Note that, for a generic method, this is the type before instantiating the method's type arguments.*
  - A **boolean** conditional expression.
- If both the second and the third operand expressions are *numeric expressions*, the conditional expression is a numeric conditional expression.
  - For the purpose of classifying a conditional, the following expressions are numeric expressions:
    - An expression of a standalone form (§15.2) with a type that is convertible to a numeric type (§4.2, §5.1.8).
    - A parenthesized numeric expression (§15.8.5).
    - A class instance creation expression (§15.9) for a class that is convertible to a numeric type.
    - A method invocation expression (§15.12) for which the chosen most specific method (§15.12.2.5) has a return type that is convertible to a numeric type.  
*Note that, for a generic method, this is the type before instantiating the method's type arguments.*
  - A numeric conditional expression.
- Otherwise, the conditional expression is a reference conditional expression. (ie a reference to a object)

## 32 while loop

```
int z = 0;
while (z < 5) {
    System.out.println(z);
    z++;
}
System.out.println("Final z = " + z);
```

### OUTPUT

```
0
1
2
3
4
Final z = 5
```

### PSEUDO-CODE

```
z=0;
START:
if (z < 5) {
    z++;
    System.out.println(z);
    END: goto START;
}
DONE: ;
```



## 33 do while loop

*Body of code is always executed at least once.* This is the major difference with the do while

```
int    z = 0;
do {
    System.out.println(z);
    z++;
} while (z < 5);
System.out.println("Final z = "+z);
```

OUTPUT

```
0
1
2
3
4
Final z = 5
```

PSEUDO-CODE

```
z=0;
START:
z++;
if (z < 5) {
    System.out.println(z);
    END: goto START;
}
DONE: ;
```

## 34 for loop

### CODE

```
int z;  
for (z = 0; z < 5; z++ ) {  
    System.out.println(z);  
}  
System.out.println("Final z = "+z);
```

### OUTPUT

```
0  
1  
2  
3  
4  
Final z = 5
```

### PSEUDO-CODE

```
int z;  
INIT: {  
    z = 0;  
}  
START:  
if (z < 5) {  
    System.out.println(z);  
    END: z++; goto START;  
}  
}  
DONE: ;
```

- Use ; to separate the three items inside the for loop
- You can declare variables in the first item. Scope is limited to the loop.

```
for (int z = 0; z < 5; z++ ) {  
    System.out.println(z);  
}
```

- Body nor third clause are not guaranteed to execute

### CODE

```
int z;  
for (z = 0; z < 5; z++ ) {  
    System.out.println(z);  
}  
System.out.println("Final z = "+z);
```

### OUTPUT

```
Final z = 0
```

- Any or all of the three clauses can be empty
- The first and thrid clauses can have mutiple statements, separate by commas

```
BasicForStatement:  
  for ( [ForInit] ; [Expression] ; [ForUpdate] ) Statement  
  
BasicForStatementNoShortIf:  
  for ( [ForInit] ; [Expression] ; [ForUpdate] ) StatementNoShortIf  
  
ForInit:  
  StatementExpressionList  
  LocalVariableDeclaration  
  
ForUpdate:  
  StatementExpressionList  
  
StatementExpressionList:  
  StatementExpression {, StatementExpression}
```

The *Expression* must have type `boolean` or `Boolean`, or a compile-time error occurs.

The scope and shadowing of a local variable declared in the *ForInit* part of a basic `for` statement is specified in §6.3 and §6.4.

- **1st clause:**
  - *LocalVariableDeclaration* (or list)  
    *or*
  - *StatementExpression* (or list)
  - NO other types of statements
- **2nd clause:**
  - must be a conditional expression (ie return a true/false).
- **3rd clause:**
  - *StatementExpression* (or list)
  - NO other types of statements

#### CODE

```
for (int r = 0, s = 1; r < 5; r++, s=2*s ) {  
    System.out.println("r="+r+", s="+s);  
}
```

#### OUTPUT

```
r=0, s=1  
r=1, s=2  
r=2, s=4  
r=3, s=8  
r=4, s=16
```

## 35 Enhanced for ("for each") loop

```
for ( VariableDeclaration : Expression) Statement
```

- variable must be declared inside the for statement parentheses.
- variable must have the same type (or compatible type such as superclass) as the array/container components
- The type of the *Expression* must be
  - a subtype of the raw type **Iterable**
  - or
  - an **array** type ([§10.1](#)),otherwise a **compile-time error** occurs.

```
int[] vals = {1,2,3};  
for (int n : vals) {    // could also define n as Integer  
    System.out.println(n);  
}
```

```
HashSet<String> myset = new HashSet<String>( Arrays.asList( new String[] { "a", "b", "c" } ) );  
for (String s : myset) {    // could also define s as Object  
    System.out.println(s);  
}
```

## 36 return, break, continue

- **return** can be placed inside any **loop**, **if**, or **switch**  
The current method is immediately exited with given return value.
- **break** can be placed inside any **loop**. or **switch**
  - The loop is exited completely: no checks, no ending clauses.
  - In terms of pseudo-code given earlier **break** becomes **goto DONE**
  - When loops are nested, the inner loop is broken out of, but not the outer loop(s)

### CODE EXAMPLE

```
int z;  
for (z = 0; z < 5; z++ ) {  
    System.out.println(z);  
    if (z==1) break;  
}  
System.out.println("Final z = "+z);
```

### OUTPUT

```
0  
1  
Final z = 1
```

- **continue** can be placed inside any **loop**
  - In terms of pseudo-code given earlier **continue** becomes **goto END**

### CODE EXAMPLE

```
for (z = 0; z < 5; z++ ) {  
    System.out.print(z);  
    if (z==1) {  
        System.out.println("*");  
        continue;  
    }  
    System.out.println("");  
}  
System.out.println("Final z = "+z);
```

### OUTPUT

```
0  
1*  
2  
3  
4  
Final z = 5
```

## CODE EXAMPLE

```
z = 0;
do {
    System.out.print(z);
    if (z++==1) {
        System.out.println("*");
        continue;
    }
    System.out.println("");
} while (z < 5);
System.out.println("Final z = "+z);
```

## OUTPUT

```
0
1*
2
3
4
Final z = 5
```

## 37 break, continue with label

- **break** and **continue** can be used with a label to break out of an entire set of nested loops, or to any of the levels in the nesting
- You can only refer to labels that are placed on the loop statements themselves. (no goto-like behavior allowed)

EXAMPLE: **break**

```
FOR1:
for (int r = 4; r < 6; r++ ) {
    System.out.println("FOR1: start");
    FOR2:
    for (int s = 0; s < r; s++ ) {
        System.out.println("FOR2: start");
        System.out.print("  r="+r+", s="+s+", r*s="+r*s);
        if (r*s==12) {
            System.out.println("  break FOR2");
            break FOR2;
        }
        System.out.println("");
    }
}
```

OUTPUT

```
FOR1: start
FOR2: start
  r=4, s=0, r*s=0
FOR2: start
  r=4, s=1, r*s=4
FOR2: start
  r=4, s=2, r*s=8
FOR2: start
  r=4, s=3, r*s=12  break FOR2
FOR1: start
FOR2: start
  r=5, s=0, r*s=0
FOR2: start
  r=5, s=1, r*s=5
FOR2: start
  r=5, s=2, r*s=10
FOR2: start
  r=5, s=3, r*s=15
FOR2: start
  r=5, s=4, r*s=20
```

EXAMPLE: **break**

```

FOR1:
for (int r = 4; r < 6; r++ ) {
    System.out.println("FOR1: start");
    FOR2:
    for (int s = 0; s < r; s++ ) {
        System.out.println("FOR2: start");
        System.out.print("  r="+r+", s="+s+", r*s="+r*s);
        if (r*s==12) {
            System.out.println("  break FOR1");
            break FOR1;
        }
        System.out.println("");
    }
}

```

```

FOR1: start
FOR2: start
  r=4, s=0, r*s=0
FOR2: start
  r=4, s=1, r*s=4
FOR2: start
  r=4, s=2, r*s=8
FOR2: start
  r=4, s=3, r*s=12  break FOR1

```

EXAMPLE: **continue**

```

FOR1:
for (int r = 4; r < 6; r++ ) {
    System.out.println("FOR1: start");
    FOR2:
    for (int s = 0; s < r; s++ ) {
        System.out.println("FOR2: start");
        System.out.print("  r="+r+", s="+s+", r*s="+r*s);
        if (r*s>=8) {
            System.out.println("  continue FOR1");
            continue FOR1;
        }
        System.out.println("");
    }
}

```

```

FOR1: start
FOR2: start
  r=4, s=0, r*s=0
FOR2: start
  r=4, s=1, r*s=4
FOR2: start
  r=4, s=2, r*s=8  continue FOR1
FOR1: start
FOR2: start
  r=5, s=0, r*s=0
FOR2: start
  r=5, s=1, r*s=5
FOR2: start
  r=5, s=2, r*s=10  continue FOR1

```





## 38 Class Inheritance

### 38.1 Root class

All classes are sub-classes of the root class named **Object**. This includes arrays as well as strings. And enums.

### 38.2 inheritance: **extends**

```
class B extends A {  
}
```

- **A inherits** from **B**.
- **B** is a **subclass** of **A**.
- **A** is a **superclass** of **B**
- If **extends** is omitted, then the class inherits directly from **Object**
- Polymorphism is not allowed: a class inherits from only from class (does not apply to interfaces)

### 38.3 this keyword

- **this** is used in instance methods to refer to the object
- **this** is used in constructors and static methods to refer to the class of the method

### 38.4 super keyword

- **super** is used in instance methods to refer to the object cast as the direct superclass of the object
- **super** is used in constructors and static methods to refer to the direct superclass of the method
- **super.super** causes a compile error

## 39 Class Constructors

- Every class must have at least one constructor defined.

```
class A {
    A() {
        out.println("executing constructor A() ");
    }
    A(String s) {
        out.println("executing constructor A(String s) "+s);
    }
}
```

- Even though its a class method, it does not have a static
- constructors have an implicit return type: the class itself. You **cannot** explicitly write the return type, nor include a **return**.
- constructors can be overloaded
- constructors are *not* inherited by subclasses
- If the default constructor, **MyClass ()**, is not explicitly defined. The following is **implicitly** definition is made.

```
class A {
    A() {} //implicit
}
```

- If you don't want to use a default constructor, define it but make it private.
- constructors can be called two ways:
  - to instantiate an object
  - as the first statement in a different constructor
- Constructors **cannot** be called in any other manner (not inside object methods or other code).
- to instantiate an object, preface a constructor call with the **new** keyword:

```
A a = new A();
A a2 = new A("hello");
```

- constructors **can** call another constructor
  - it **must** be the first statement in the constructor
  - it can be a constructor of the **direct superclass** using **super**
  - it can be different constructor of the same class using **this**. but eventually in the chain of calls there must be a call to a constructor of the **direct superclass**
  - if no call to another constructor is given explicitly, then an *implicit call* to **super ()** is made as the first statement

```
class A {
    A() {}
    A(Double d) {
        out.println("d "+d);
    }
}

class B extends A {
```

```
B() {  
    super(23.2);  
}  
B(String s) {  
    super(); //implicit  
    out.println("executing constructor B(String s) "+s);  
}  
B(Integer n) {  
    this(); // calls B()  
    out.println("executing constructor B(Integer n) "+n);  
}  
}
```

- Note that constructors are not inherited. so `new B(double)` will cause a compile error

## 40 Class inheritance examples

```
package ron.inheritance;

import static java.lang.System.out;

class A {
    A() {
        out.println("executing constructor A() ");
    }
    A(String label) {
        out.println("executing constructor A(String label) for: "+label);
    }
    String getClassName() {
        return "A";
    }
}

class B extends A {
    B(String label) {
        // implicit call of constructor super.() occurs here
        out.println("executing constructor B(String label) for: "+label);
    }
    @Override
    String getClassName() {
        return "B";
    }
}

class C extends A {
    C(String label) {
        super(label); // explicit call of constructor. must be first statement and only constructor
        call!
        out.println("executing constructor C(String label) for: "+label);
    }
    @Override
    String getClassName() {
        return "C";
    }
}

class D extends C {
    D(String label) {
        super(label); // note super.super() won't compile
        out.println("executing constructor D(String label) for: "+label);
    }
    @Override
    String getClassName() {
        return "D";
    }
}

class F extends A{
    // no constructor defined so a default will be created that does nothing except call the
    super.()
}
```

```
class G extends A{
    G() {
        out.println("executing constructor G() ");
    }
    G(String label) {
        this();
        out.println("executing constructor G(String label) for: "+label);
    }
}

class Inheritance {

    public static void main(String[] args) {

        out.println("A a1 = new A();");
        A a1 = new A();
        out.println("a1.getClassName() = "+a1.getClassName());
        out.println("");

        out.println("A a2 = new A(\"a2\");");
        A a2 = new A("a2");
        out.println("a2.getClassName() = "+a2.getClassName());
        out.println("");

        out.println("B b1 = new B(\"b1\");");
        B b1 = new B("b1");
        out.println("b1.getClassName() = "+b1.getClassName());
        out.println("");

        out.println("B b2 = new B(\"b2\");");
        try {
            B b2 = (B) new A(); // narrowing here will cause runtime exception
            out.println("b2.getClassName() = "+b2.getClassName());
        } catch (java.lang.ClassCastException e) {
            out.println("Exception occurred and caught: "+e);
        } finally {
            out.println("");
        }

        out.println("C c1 = new C(\"c1\");");
        C c1 = new C("c1");
        out.println("c1.getClassName() = "+c1.getClassName());
        out.println("");

        out.println("D d1 = new D(\"d1\");");
        D d1 = new D("d1");
        out.println("d1.getClassName() = "+d1.getClassName());
        out.println("");

        out.println("A d2 = d1;");
        A d2 = d1; // widening, no cast needed
        out.println("d2.getClassName() = "+d2.getClassName());
        out.println("");

        out.println("A d3 = (A) d1;");
        A d3 = (A) d1; // widening with explicit cast
        out.println("d3.getClassName() = "+d3.getClassName());
        out.println("");

        out.println("F f1 = new F(\"f1\");");
        F f1 = new F();
    }
}
```

```
    out.println("f1.getClassName() = "+f1.getClassName());
    out.println("");

    out.println("G g1 = new G(\"g1\");");
    G g1 = new G("g1");
    out.println("g1.getClassName() = "+g1.getClassName());
    out.println("");
}
}
```

The above produces the following output

```
A a1 = new A();
executing constructor A()
a1.getClassName() = A

A a2 = new A("a2");
executing constructor A(String label) for: a2
a2.getClassName() = A

B b1 = new B("b1");
executing constructor A()
executing constructor B(String label) for: b1
b1.getClassName() = B

B b2 = new B("b2");
executing constructor A()
Exception occurred and caught: java.lang.ClassCastException: class
ron.inheritance.A cannot be cast to class ron.inheritance.B (ron.inheritance.A and
ron.inheritance.B are in unnamed module of loader 'app')

C c1 = new C("c1");
executing constructor A(String label) for: c1
executing constructor C(String label) for: c1
c1.getClassName() = C

D d1 = new D("d1");
executing constructor A(String label) for: d1
executing constructor C(String label) for: d1
executing constructor D(String label) for: d1
d1.getClassName() = D

A d2 = d1;
d2.getClassName() = D

A d3 = (A) d1;
d3.getClassName() = D

F f1 = new F("f1");
executing constructor A()
f1.getClassName() = A

G g1 = new G("g1");
executing constructor A()
executing constructor G()
executing constructor G(String label) for: g1
g1.getClassName() = A
```

# 41 Class members

## 41.1 members

The members of a class are its fields and methods.

That is, a member is a general term referring to field or a method.

## 41.2 fields

Variable inside the body of a class (ie not inside a method) are called fields.

**static fields** are class variables

**non-static fields** are instance variables.

## 41.3 methods

Functions inside the body of a class are called **methods**.

**static methods** are class methods

**non-static methods** are instance methods.

## 41.4 Rules

- Fields can have same name in different levels of the hierarchy.  
The types can be same or different
- Methods can have same signature (name and list of parameter types) in different levels of the hierarchy.
- Class fields cannot share same name as instance fields (at any level of the hierarchy).
- Class methods cannot share same signature as instance methods (at any level of the hierarchy).

## 41.5 Hiding fields

Identically named fields at different levels of the hierarchy can have the same or different types

## 41.6 Class methods

Class methods with same signature at different levels of the hierarchy can have same or different return types.

## 41.7 Instance method return types and covariant return types

An overriding method must have either:

- the same return type as the overridden method
- a subtype of return type as the overridden method

The later is called a **covariant return** type.

## 41.8 Overriding and Hiding

The following table summarizes how Java chooses which definition to choose.

<b>class fields</b>	hiding	
<b>class methods</b>	hiding	
<b>instance fields</b>	hiding	
<b>instance methods</b>	called via <b>super.</b>	hiding
	otherwise	<b>overriding</b>

**Hiding** chooses from the **compile-time class**.

**Overriding** chooses starting at the **runtime class**.



## 41.9 Run-time class and Compile-time class

The **runtime class** is the actual class of an object instance, ie the class of the constructor that was used to create the object.

Definition of **compile-time class**

	called via	compile-time class
class members	<code>B.member</code>	B
	<code>member</code> , inside class method of <code>B</code>	B
	<code>((B) object).member</code>	B
	<code>member</code> inside instance method of class <code>B</code>	B
	<code>this.member</code> inside instance method of class <code>B</code>	B
	<code>super.member</code> inside instance method of class <code>B</code>	A
instance members	<code>((B) object).member</code>	B
	<code>member</code> inside instance method of class <code>B</code>	B
	<code>this.member</code> inside instance method of class <code>B</code>	B
	<code>super.member</code> inside instance method of class <code>B</code>	A

where

- The runtime class of object/instance is `R`.
- `R` is a subclass of `B`, which is a direct subclass of `A`.
- Shaded cells cause a compile-time warning.

## 42 Access Modifiers

- The **private** access modifier means that *only code inside the class itself* can access this Java field.
- The (*package-private*) access modifier means that only code inside the class itself, or other classes in the same package, can access the field. **You don't and can't actually write the package modifier.** By leaving out any access modifier, the access modifier defaults to package scope. *Not subclasses unless in same package!*
- The **protected** access modifier is like the package modifier, except *subclasses of the class can also access the field, even if the subclass is not located in the same package.*
- The **public** access modifier means that the field can be accessed by all classes in your application.

Access level modifiers determine whether other classes can use a particular field or invoke a particular method. There are two levels of access control:

- At the top level—**public**, or *package-private* (no explicit modifier).
- At the member level—**public**, **private**, **protected**, or *package-private* (no explicit modifier).

A class may be declared with the modifier **public**, in which case that class is visible to all classes everywhere. If a class has **no modifier** (the default, also known as **package-private**), it is visible only within its own package (packages are named groups of related classes — you will learn about them in a later lesson.)

At the member level, you can also use the **public** modifier or **no modifier** (*package-private*) just as with top-level classes, and with the same meaning. For members, there are two additional access modifiers: **private** and **protected**.

The **private** modifier specifies that the member can only be accessed in its own class. The **protected** modifier specifies that the member can only be accessed within its own package (as with *package-private*) and, in addition, by a subclass of its class in another package.

The following table shows the access to members permitted by each modifier.

Access Levels				
Modifier	Class	Package	Subclass in another package	World
<b>public</b>	Y	Y	Y	Y
<b>protected</b>	Y	Y	Y	N
( <i>package-private</i> )	Y	Y	N	N
<b>private</b>	Y	N	N	N

The first data column indicates whether the class itself has access to the member defined by the access level. As you can see, a class always has access to its own members. The second column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member. The third column indicates whether subclasses of the class declared outside this package have access to the member. The fourth column indicates whether all classes have access to the member.

Access levels affect you in two ways. First, when you use classes that come from another source, such as the classes in the Java platform, access levels determine which members of those classes your own classes can use. Second, when you write a class, you need to decide what access level every member variable and every method in your class should have.

## 43 Static Fields: Access by Class Name

You can always access a static field of a class directly by the class name anywhere.

Static fields implement **inheritance with hiding**.

- Static fields are **inherited** by subclasses.
- **Hiding** is used to resolve a static field with same name as inherited static field  
**Hiding** means that A and B each have a separate independent variable named "strAB".
- Not only can A and B have static fields of the same name, the type can be different (even primitives versus objects)

```
class A {  
    static String strA = "A.strA";  
    static String strAB = "A.strAB";  
    static String x = "Hello";  
}  
  
class B extends A {  
    static String strB = "B.strB";  
    static String strAB = "B.strAB";  
    static int x = 10;  
}
```

```
→A.strA = A.strA  
→A.strAB = A.strAB  
→A.x = Hello  
  
→B.strA = A.strA  
→B.strAB = B.strAB // B has its own independent variable strAB  
→B.strB = B.strB  
→B.x = 10
```

- B **inherits** field strA from A
- B.strAB **hides** A.strAB

## 44 Static Fields: Accessing from Class Instances

(All of these were run from outside the class hierarchy of the declaring class.)

- **Hiding:** Access to static fields from an object goes to the field of the *object's declared type* (not runtime type)!
- The compiler warns that this is a dangerous practice.
  - Best practice is to implement instance getter method(s)

```
A a = new A();
out.println(" → a.strA = " + a.strA); // compiler warning
out.println(" → a.strAB = " + a.strAB); // compiler warning
```

```
→ a.strA = A.strA
→ a.strAB = A.strAB
```

```
B b = new B();
out.println(" → b.strA = " + b.strA); // compiler warning
out.println(" → b.strAB = " + b.strAB); // compiler warning
out.println(" → b.strB = " + b.strB); // compiler warning out.println("");
```

```
→ b.strA = A.strA
→ b.strAB = B.strAB
→ b.strB = B.strB
```

Same object, but variable referencing it is a different type

```
A b_as_A = (A) b;
out.println(" → b_as_A.getClass() = " + b_as_A.getClass()); // run-time class of object
out.println(" → b_as_A.strA = " + b_as_A.strA); // compiler warning
out.println(" → b_as_A.strAB = " + b_as_A.strAB); // compiler warning
✗ out.println("b_as_A.strAB = " + b_as_A.strB); // compiler error
```

```
→ b_as_A.getClass() = class ron.staticinheritance.B
→ b_as_A.strA = A.strA
→ b_as_A.strAB = A.strAB
```

```
Object b_as_Object = (Object) b;
out.println(" → b_as_A.getClass() = " + b_as_Object.getClass()); // run-time class of
object
✗ out.println(" → b_as_A.strA = " + b_as_Object.strA); // compiler error
```

```
→ b_as_A.getClass() = class ron.staticinheritance.B
```

## 45 Static Methods

Accessing static methods by class name follows same rules as static fields.

You can always access a static method of a class directly by the class name, anywhere.

- Static methods implement **inheritance with hiding**
- **Hiding** is used to resolve a static method with same **signature** (name + argument list, see later section) as an inherited static method  
**Hiding** means that A and B each have a separate independent methods named "getAB".
- **Hiding** means that the method is resolved by the declared class of the instance variable/expression, not run-time class of the object itself.
- **X this** and **super** are *not* allowed inside static methods, ie cause compile errors (only allowed in instance methods and on first line of constructor)
- Static methods have no knowledge of instance fields and methods

```
class A {
    static String strA = "A.strA";
    static String strAB = "A.strAB";

    static String getX()
    {
        return "X from class A";
    }

    static String getY()
    {
        return "Y from class A";
    }
}

class B extends A {
    static String strB = "B.strB";
    static String strAB = "B.strAB";

    static String getY()
    {
        return "Y from class B";
    }

    static String getZ()
    {
        return "Z from class B";
    }
}
```

```
→A.getX() = X from class A
→A.getY() = Y from class A

→B.getX() = X from class A (inheritance)
→B.getY() = Y from class B (hiding/independence)
→B.getZ() = Z from class B
```

## 46 Accessing static fields/methods from static methods

**Hiding:** resolution depends on *the class of the calling method*:

- Inside a method of class *X*, any access to a static field *fieldname* is same as *X.fieldname*
- Inside a method of class *X*, any access to a static method *methname* is same as *X.methname*

After this replacement, the rules of inheritance with hiding apply.

Here is a tricky case. Originally `B.getZ()` is called, but `getZ()` is inherited from `A`, inside `A.getZ()` refers to `Z`, its own `Z`.

```
class A {  
    static String Z = "A.Z";  
  
    static String getZ()  
    {  
        return Z;  
    }  
}  
  
class B extends A {  
    static String Z = "B.Z";  
}
```

```
→A.Z = A.Z  
→B.Z = B.Z  
  
→A.getZ() = A.Z  
→B.getZ() = A.Z
```

## 47 Accessing static fields/methods from instance methods

**Hiding:** It only depends on *the class of the calling method*:

- Inside a method of class X, any access to a static field *fieldname* is same as *X.fieldname*
- Inside a method of class X, any access to a static method *methname* is same as *X.methname*
- keywords **this** (same as direct reference) and **super** can be used for static members when inside instance methods, **use with care**

After this replacement, the rules of inheritance with hiding apply.

### EXAMPLE

```
class A {
    static String Z = "A.Z";
    static String Y = "A.Y";

    String getZ()
    {
        return Z;
    }
    String getY()
    {
        out.println("A instance.getY(): Executing");
        return Y;    // return this.Y has same effect but generates warning
    }
    String getY2()
    {
        out.println("A instance.getY2(): Executing");
        this.return Y;    // same effect but generates compiler warning
    }
}

class B extends A {
    static String Z = "B.Z";
    static String Y = "B.Y";

    String getY()
    {
        out.println("B instance.getY(): Executing");
        return super.getY();
    }
}
```

```

public static void main(String[] args) {

    A a = new A();
    printObject(" →a.Z", a.Z);           // compiler warning
    printObject(" →a.getZ()", a.getZ());
    printObject(" →a.getY()", a.getY());
    printObject(" →a.getY2()", a.getY2());
    out.println("");

    B b = new B();
    printObject(" →b.Z", b.Z);           // compiler warning
    printObject(" →b.getZ()", b.getZ());
    printObject(" →b.getY()", b.getY());
    out.println("");

    A basA = b;
    printObject(" →basA.Z", basA.Z);     // compiler warning
    printObject(" →basA.getZ()", basA.getZ());
    printObject(" →basA.getY()", basA.getY());
    out.println("");

}

```

```

→a.Z = A.Z
→a.getZ() = A.Z

A instance.getY(): Executing
→a.getY() = A.Y

A instance.getY2(): Executing
→a.getY2() = A.Y


→b.Z = B.Z
→b.getZ() = A.Z

B instance.getY(): Executing
A instance.getY(): Executing
→b.getY() = A.Y


→basA.Z = A.Z
→basA.getZ() = A.Z

B instance.getY(): Executing
A instance.getY(): Executing
→basA.getY() = A.Y

```



## 48 Instance (ie non-static) Fields

Instance fields implement **inheritance with hiding** (different from instance methods!)

- instance fields are **inherited** by subclasses.

**Instance fields** can be accessed in two ways

- by variable name (in any code)
- **hiding** means that name resolution depends on *declared class of variable*

```
Object obj;  
out.println( obj.field );
```

- via **this.member** and **super.member** (when inside an instance method)
- the **this** keyword can be omitted, but use care because of clashes with local variables
- **hiding** means that name resolution depends on the *class of the instance method being executed* (not run-time class of the object)

```
out.println( this.field );  
out.println( super.field );
```

## 49 Instance (ie non-static) Methods

Instance methods implement **inheritance with overriding**

- instance fields are **inherited** by subclasses.
- methods are only considered to be the same if they have the same **signature** (name + argument list, see later section)
- For clarity use the optional **@Override annotation** for a method that overrides a method from a superclass. Note the capital O in **@Override**

**Instance methods** can be accessed in two ways

- by `variableName.method()` (in any code)
- **overriding** means that name resolution depends on *run-time class of the object*

```
Object obj;  
out.println( obj.method() );
```

- via `this.method()` and `super.method()` (when inside an instance method)
- the `this` keyword can be omitted
- **overriding** means that name resolution depends on *run-time class of the object* (regardless of which calling method and where the calling method is defined)

```
out.println( this.method() );  
out.println( super.method() );
```

## 50 The definition of this

This refers to the **current object**, `obj`, cast as the class of where the code resides, the **residing class** `R`.

```
this = (R) obj;
```

Let the run-time class of `obj` be `T`, where `T` is either the same as `R` or a subclass of `R`.

The rules have subtleties:

- **Hiding:** For fields, only fields defined in `R` (and those inherited from the superhierarchy) can be accessed.
  - Identically named field(s) in the subhierarchy (down to `T`) are ignored and cannot be accessed
- **Overriding:** For methods, only methods defined in `R` (and those inherited from the superhierarchy) can be accessed.
  - However, the subhierarchy (down to `T`) is searched for overriding methods. The method invoked is the method that would be invoked for `(T) obj`.

### 15.8.3. `this`

The keyword `this` may be used only in the following contexts:

- in the body of an instance method or default method ([§8.4.7](#), [§9.4.3](#))
- in the body of a constructor of a class ([§8.8.7](#))
- in an instance initializer of a class ([§8.6](#))
- in the initializer of an instance variable of a class ([§8.3.2](#))
- to denote a receiver parameter ([§8.4](#))

If it appears anywhere else, a compile-time error occurs.

The keyword `this` may be used in a lambda expression only if it is allowed in the context in which the lambda expression appears. Otherwise, a compile-time error occurs.

When used as a primary expression, the keyword `this` denotes a value that is a reference to the object for which the instance method or default method was invoked ([§15.12](#)), or to the object being constructed. The value denoted by `this` in a lambda body is the same as the value denoted by `this` in the surrounding context.

The keyword `this` is also used in explicit constructor invocation statements ([§8.8.7.1](#)).

The type of `this` is the class or interface type `T` within which the keyword `this` occurs.

Default methods provide the unique ability to access `this` inside an interface. (All other interface methods are either **abstract** or **static**, so provide no access to `this`.) As a result, it is possible for `this` to have an interface type.

At run time, the class of the actual object referred to may be `T`, if `T` is a class type, or a class that is a subtype of `T`.

## 51 The exact definition of super

Let the class of where the code resides, the **residing class**, be **R**.

Let the superclass of **R** be **Q**.

Let the run-time class of the **current object**, **obj**, be **T**, where **T** is either the same as **R** or a subclass of **R**.

Super refers to the **current object**, **obj**, cast as **Q**

```
super = (Q) obj;
```

The rules are different (and simpler) than those for **this**, namely methods are invoked via hiding, not overriding.

- **Hiding:** For fields, only fields defined in **Q** (and those inherited from the superhierarchy) can be accessed.
  - Identically named field(s) in the subhierarchy (down to **T**) are ignored and cannot be accessed
- **Hiding:** For methods, only methods defined in **Q** (and those inherited from the superhierarchy) can be accessed.
  - A Method which has an identical **signature** (name + argument list, see later section) method(s) in the subhierarchy (down to **T**) are ignored and cannot be accessed

### 15.11.2. Accessing Superclass Members using **super**

The form **super**. *Identifier* refers to the field named *Identifier* of the current object, but with the current object viewed as an instance of the superclass of the current class.

The form **T**. **super**. *Identifier* refers to the field named *Identifier* of the lexically enclosing instance corresponding to **T**, but with that instance viewed as an instance of the superclass of **T**.

The forms using the keyword **super** are valid only in an instance method, instance initializer, or constructor of a class, or in the initializer of an instance variable of a class. If they appear anywhere else, a compile-time error occurs.

These are exactly the same situations in which the keyword **this** may be used in a class declaration ([§15.8.3](#)).

It is a compile-time error if the forms using the keyword **super** appear in the declaration of class `Object`, since `Object` has no superclass.

Suppose that a field access expression **super**. *f* appears within class **C**, and the immediate superclass of **C** is class **S**. If *f* in **S** is accessible from class **C** ([§6.6](#)), then **super**. *f* is treated as if it had been the expression **this**. *f* in the body of class **S**. Otherwise, a compile-time error occurs.

Thus, **super**. *f* can access the field *f* that is accessible in class **S**, even if that field is hidden by a declaration of a field *f* in class **C**.

**Example 15.12.4.4-2. Method Invocation Using super**

An overridden instance method of a superclass may be accessed by using the keyword *super* to access the members of the immediate superclass, bypassing any overriding declaration in the class that contains the method invocation.

When accessing an instance variable, *super* means the same as a cast of *this* (§15.11.2), but this equivalence does not hold true for method invocation. This is demonstrated by the example:

```
class T1 {
    String s() { return "1"; }
}
class T2 extends T1 {
    String s() { return "2"; }
}
class T3 extends T2 {
    String s() { return "3"; }
    void test() {
        System.out.println("s()=\t\t"      + s());
        System.out.println("super.s()=\t"  + super.s());
        System.out.println("((T2)this).s()=\t" + ((T2)this).s());
        System.out.println("((T1)this).s()=\t" + ((T1)this).s());
    }
}
class Test2 {
    public static void main(String[] args) {
        T3 t3 = new T3();
        t3.test();
    }
}
```

which produces the output:

```
s()=          3
super.s()=     2
((T2)this).s()= 3
((T1)this).s()= 3
```

The casts to types *T1* and *T2* do not change the method that is invoked, because the instance method to be invoked is chosen according to the run-time class of the object referred to by *this*. A cast does not change the class of an object; it only checks that the class is compatible with the specified type.

## 52 Instance Field and Method Example

```
class A {
    String x = "A.x";
    String y = "A.y";
    String z = "A.z";

    String getX()
    {
        out.println("Executing: A instance.getX()");
        return x;
    }
    String getX2()
    {
        out.println("Executing: A instance.getX2()");
        return getX();
    }
    String getY()
    {
        out.println("Executing: A instance.getY()");
        return y;
    }
    String getZ()
    {
        out.println("Executing: A instance.getZ()");
        return z;
    }
}

class B extends A {
    String x = "B.x";
    String y = "B.y";
    String z = "B.z";

    String getX()
    {
        out.println("Executing: B instance.getX()");
        return x;
    }
    String getY()
    {
        out.println("Executing: B instance.getY()");
        return super.getY();
    }
}
```

```
public static void main(String[] args) {  
  
    A a = new A();  
    printObject(" →a.x", a.x);  
    printObject(" →a.getX()", a.getX());  
    printObject(" →a.getX2()", a.getX2());  
    printObject(" →a.y", a.y);  
    printObject(" →a.getY()", a.getY());  
    printObject(" →a.z", a.z);  
    printObject(" →a.getZ()", a.getZ());  
    out.println("");  
  
    B b = new B();  
    printObject(" →b.x", b.x);  
    printObject(" →b.getX()", b.getX());  
    printObject(" →b.getX2()", b.getX2());  
    printObject(" →b.y", b.y);  
    printObject(" →b.getY()", b.getY());  
    printObject(" →b.z", b.z);  
    printObject(" →b.getZ()", b.getZ());  
    out.println("");  
  
    A basA = b;  
    printObject(" →basA.x", basA.x);  
    printObject(" →basA.getX()", basA.getX());  
    printObject(" →basA.getX2()", basA.getX2());  
    printObject(" →basA.y", basA.y);  
    printObject(" →basA.getY()", basA.getY());  
    printObject(" →basA.z", basA.z);  
    printObject(" →basA.getZ()", basA.getZ());  
    out.println("");  
  
}
```

```
→a.x = A.x
Executing: A instance.getX()
→a.getX() = A.x
Executing: A instance.getX2()
Executing: A instance.getX()
→a.getX2() = A.x

→a.y = A.y
Executing: A instance.getY()
→a.getY() = A.y

→a.z = A.z
Executing: A instance.getZ()
→a.getZ() = A.z

→b.x = B.x
Executing: B instance.getX()
→b.getX() = B.x
Executing: A instance.getX2()
Executing: B instance.getX()
→b.getX2() = B.x

→b.y = B.y
Executing: B instance.getY()
Executing: A instance.getY()
→b.getY() = A.y

→b.z = B.z
Executing: A instance.getZ()
→b.getZ() = A.z

→basA.x = A.x
Executing: B instance.getX()
→basA.getX() = B.x
Executing: A instance.getX2()
Executing: B instance.getX()
→basA.getX2() = B.x

→basA.y = A.y
Executing: B instance.getY()
Executing: A instance.getY()
→basA.getY() = A.y

→basA.z = A.z
Executing: A instance.getZ()
→basA.getZ() = A.z
```



## 53 Method Signature and Overloading

### 53.1 Definitions

The **signature** of a method is the name and the list of parameter types.

The **return type** of a method is the type that is returned.

#### 53.1.1 Example 1

method	<pre>void getX(void) {     return; }</pre>
signature	( getX, void )
return type	void

#### 53.1.2 Example 2

method	<pre>void getX() {     return; }</pre>
signature	( getX, void )
return type	void

#### 53.1.3 Example 3

method	<pre>int getX() {     return 1; }</pre>
signature	( getX, void )
return type	int

#### 53.1.4 Example 4

method	<pre>void getX(int n) {     return; }</pre>
signature	( getX, int )
return type	void

#### 53.1.5 Example 5

method	<pre>Number getX(int n, String s) {     return 0.0; }</pre>
signature	( getX, int, String )
return type	Number

#### 53.1.6 Example 6

method	<pre>Number getX(String s, int n) {     return 0.0; }</pre>
signature	( getX, String, int )
return type	Number

## 53.2 Method Overloading

Overloading a method means having two or more methods in a class (or class hierarchy) with the same name, but different signature.

Each overloaded method is treated as a different method.

The return types are irrelevant for overloading. They can be the same or different.

This is in contrast to return types for overriding methods.

## 53.3 Operators can never be overridden

This is in contrast to C++.

## 53.4 The operator == is never overridden and cannot be overridden

For primitive variables the operator `==` returns true iff the literal values are equal

For reference variables the operator `==` returns true iff the object identifiers are equal (ie they refer to the same object).

For function equality, objects need to override the `compareTo()` method inherited from `Object`.

## 53.5 Using `compareTo()`

If you override `compareTo()`, you MUST override `hashCode()`

TODO: add details of this

## 53.6 Comparable

To support greater than and less than for an object, you need to implement the Interface `java.lang.Comparable<T>`.

```
Interface java.lang.Comparable<T> {  
    int compareTo(T o);  
}
```

This interface contains a single function which compares this object with the specified object and returns

- a negative integer if this object is **less than** the specified object ◦
- zero if this object is **equal to** the specified object ◦
- a positive integer if this object is **greater than** the specified object ◦

Example stub

```
class C implements Comparable<C> {  
    @Override  
    public int compareTo(C o) {  
        // TODO Auto-generated method stub  
        return 0;  
    }  
}
```

**int** compareTo(C o)

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

- The implementor must ensure  $\text{sgn}(x.\text{compareTo}(y)) == -\text{sgn}(y.\text{compareTo}(x))$  for all  $x$  and  $y$ . (This implies that  $x.\text{compareTo}(y)$  must throw an exception iff  $y.\text{compareTo}(x)$  throws an exception.)
- The implementor must also ensure that the relation is transitive:  $(x.\text{compareTo}(y) > 0 \ \&\& \ y.\text{compareTo}(z) > 0)$  implies  $x.\text{compareTo}(z) > 0$ .
- Finally, the implementor must ensure that  $x.\text{compareTo}(y) == 0$  implies that  $\text{sgn}(x.\text{compareTo}(z)) == \text{sgn}(y.\text{compareTo}(z))$ , for all  $z$ .
- It is strongly recommended, but *not* strictly required that  $(x.\text{compareTo}(y) == 0) == (x.\text{equals}(y))$ . Generally speaking, any class that implements the `Comparable` interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

In the foregoing description, the notation  $\text{sgn}(\text{expression})$  designates the mathematical *signum* function, which is defined to return one of  $-1$ ,  $0$ , or  $1$  according to whether the value of *expression* is negative, zero or positive.

- There also exists an interface called `Comparator<T>` which is used for sorting the elements in a collection. Be care not to confuse these.

## 54 Static Field Initialization and Initialization Blocks

### 54.1 initialization and constructors

Not sure if this is true, but I apparently encountered problems at one point.

- Be sure to include all static variable initialization and all static code blocks *before all constructors*.
- Otherwise they might not get executed before constructing instances
- Not following this rule may not cause a compile-time error so be careful

I like to follow the rule to place all static members first, before constructors and instance members.

### 54.2 Default initialization

- A static field that is declared but not explicitly initialized is given default initialization.
- This is same as array element initialization
  - 0 for numeric values
  - **null** for objects (including strings)
  - **false** for booleans
- However, keep in mind that this is different from local variables, which are not initialized

### 54.3 Instances

- static fields initialization cannot instantiate an object or reference any instance methods

### 54.4 Class field initialization using static code blocks

- static fields can be initialized in static code blocks
  - static code blocks can be placed before or after the field definition
  - order matters!

#### 54.4.1 Example

```
public class FieldInit {  
  
    static { x = 1; }  
    static int x;  
  
    static { y = 1; }  
    static int y = 2;  
  
    static { z = 1; }  
    static int z = 2;  
    static { z = 3; }  
  
    public static void main(String[] args) {  
        out.println("x = "+x);  
        out.println("y = "+y);  
        out.println("z = "+z);  
    }  
}
```

#### OUTPUT

```
x = 1  
y = 2  
z = 3
```

#### 54.4.2 ❌ Class fields in expression before declaration referred to via *field*

You cannot refer to a field via its simple name that has not yet been declared.  
You can only use a field via its simple name to assign it a value.

```
static { u = v; } ❌ compile-time error
static int u;
static int v;
```

```
static {
    v = 15;
    int x = v * 13; ❌ compile-time error
}
static int v;
```

However, you can get around this rule by referencing using the full name!

#### 54.4.3 ✅ Class fields in expression before declaration referred to via *MyClass.field*

You can use fields in expression if you use the fullname!

```
static {
    out.println("v = " + FieldInit.v);
    u = FieldInit.v + 2;
    out.println("u = " + FieldInit.u);
}
static int u;
static int v;
```

#### Output

```
v = 0    // default initialization
u = 2
```

### 54.5 Static blocks can contain local variables and code

#### 54.5.1 Example

```
static {
    int x1 = 0;
    int x2 = 23;
    while (x1+x2 < 100) {
        x1 = x1 + x2;
    }
    a = x1;
}
static int a;
```

#### OUTPUT

```
a = 92
```

## 54.6 class fields initialization expression references

You can use other class fields in direct initializations.

### 54.6.1 Example

```
static int b = 7;  
static int c = 2*b;
```

#### OUTPUT

```
c = 14
```

### 54.6.2 Example

```
static int b;  
static int c = b + 1;
```

#### OUTPUT

```
b = 0 // default init value  
c = 1
```

### 54.6.3 ❌ Non-Example (simple class field name)

You can't refer to a field via simple name that is declared afterword

```
static int c = 2*b; ❌ compile-time error  
static int b = 7;
```

### 54.6.4 ✅ Example (full class field name)

You can refer to a field via its full name, even if it is declared afterword

```
static int c = FieldInit.b + 1;  
static int b = 7 ;
```

#### OUTPUT

```
b = 7  
c = 1 // c used default init value for b
```

## 54.7 class fields can be initialized using static methods

- It is irrelevant where the static method is defined
- you can use simple names.

- Be careful though, as code below demonstrates, order of variable initialization and blocks is relevant .

```
static int f = 21;
static int g = init();
static { f = 100; }
static int h = init();

static int init() {
    return f+1;
}
```

```
f = 100
g = 22
h = 101
```

### 54.7.1 Use of final with class fields

A class field can only be final if it is *initialized at declaration*.

#### EXAMPLE

```
public class FieldInit {
    //explicit init in this manner is the only valid way to set a final static field
    final static int x = 1;
}
```

- A final static field is a true constant

### 54.8 class variables and instantiation

- a class variable can be initialized to an instance of the same class or an expression thereof
  - if a constructor refers to a class variable before it is initialized, the constructor will get the default value!
  - so be careful
- static blocks and static methods can also create instances.
  - same warning applies

## 55 Instance Field Initialization and Initialization Blocks

### 55.1 Default initialization

- A field that is declared but not explicitly initialized is given default initialization.
- This is same as array element initialization
  - 0 for numeric values
  - **null** for objects (including strings)
  - **false** for booleans
- However, keep in mind that this is different from local variables, which are not initialized

### 55.2 instance field initialization using code blocks

- fields can be initialized in code blocks
  - code blocks can be placed before or after the field definition
  - order matters!

#### 55.2.1 Example

```
public class FieldInit {  
  
    int w;  
  
    { x = 1; }  
    int x;  
  
    {  
        y = 1;  
    }  
    int y = 2;  
  
    {  
        z = 1;  
    }  
    int z = 2;  
    {  
        z = 3;  
    }  
  
    FieldInit() {  
  
    }  
  
    public static void main(String[] args) {  
  
        FieldInit obj = new FieldInit();  
        out.println("obj.w = "+ obj.w);  
        out.println("obj.x = "+ obj.x);  
        out.println("obj.y = "+ obj.y);  
        out.println("obj.z = "+ obj.z);  
  
    }  
}
```



## OUTPUT

```
obj.w = 0
obj.x = 1
obj.y = 2
obj.z = 3
```

### 55.2.1.1 Use of final

An instance field without initialization cannot be final. Default initialization cannot be applied to final variables.

```
public class FieldInit {
    final int w; ✗ compile-time error

    FieldInit() {
    }
}
```

An final instance field must be initialized

- at declaration

OR

- in every constructor

OR

- in a single block (before or after)

## EXAMPLE

```
public class FieldInit {

    final int u = 23;

    final int w;

    { x = 1; }
    final int x;

    final int y;
    { y = 1; }

    FieldInit() {
        w = 1;
    }
}
```

### 55.2.2 ✗ Instance Fields in expression before declaration

You cannot refer (in an expression) to an instance field that has not yet been declared.

You can only use an instance field via its simple name to assign it a value.

```
{ u = v; } ✗ compile-time error
int u;
int v;
```

```
{  
    v = 15;  
    int x = v * 13; ✗ compile-time error  
}  
int v;
```

Unlike class fields, there is no way around this.

## 55.3 Instance blocks can contain local variables and code

### 55.3.1 Example

```
{  
    int x1 = 0;  
    int x2 = 23;  
    while (x1+x2 < 100) {  
        x1 = x1 + x2;  
    }  
    a = x1;  
}  
int a;
```

#### OUTPUT

```
obj.a = 92
```

## 55.4 Instance fields initialization expression references

You can use other instance fields in direct initializations, as long as the referenced field has already been declared.

### 55.4.1 Example

```
int b = 7;  
int c = 2*b;
```

#### OUTPUT

```
obj.b = 7  
obj.c = 14
```

#### 55.4.1.1 Use of Final

The fields `b` and `c` can both be declared as final.

```
final int b = 7;  
final int c = 2*b;
```

### 55.4.2 Example

```
int b;  
int c = b + 1;
```

## OUTPUT

```
obj.b = 0 // default init value
obj.c = 1
```

### 55.4.2.1 Use of Final

The field `b` *cannot* be declared as final.

The field `c` can also be declared final.

```
final int b; ✗ compile-time error // cannot declare uninitialized field as final
final int c = b + 1; ✓ // this is ok
```

### 55.4.3 ✗ Non-Example (instance field name)

You can't refer to a instance field that is declared afterword

```
int c = 2*b; ✗ compile-time error
int b = 7;
```

Unlike class fields, there is no way around this.

## 55.5 Instance fields can be initialized using class fields

When initializing using a class field, the value at the time of instantiation is used.

```
public class FieldInit {

    static int A = 1;

    int x = A;

    FieldInit() {
    }

    public static void main(String[] args) {
        FieldInit.A = 2;
        FieldInit obj = new FieldInit();
        FieldInit.A = 3;
        out.println("obj.x = "+ obj.x);
    }
}
```

## OUTPUT

```
obj.x = 2
```

### 55.5.1 Use of Final

The field `x` can also be declared final

```
final int x = A;
```

## 55.6 Instance fields can be initialized using class methods

```
static int sinit() {  
    return 13;  
}  
  
int x = sinit();  
  
FieldInit() {  
}
```

### OUTPUT

```
obj.x = 13
```

## 55.7 Instance fields and constructors

- Constructors are run *after* variables are declared and initialized and *after* all code blocks are run.
- It is irrelevant where the constructor is located.

### EXAMPLE

```
{ x = 1; }  
int x = 2;  
FieldInit() {  
    x = 3;  
}
```

### OUTPUT

```
obj.x = 3
```

### EXAMPLE

```
FieldInit() {  
    x = 3;  
}  
{ x = 1; }  
int x = 2;
```

### OUTPUT

```
obj.x = 3
```

## 56 Local Variables

- local variables must be explicitly initialized
  - can be initialized at declaration
  - can be initialized later in code
  - trying to use an uninitialized local will cause a **compile-time error**
- local variables can be defined anywhere inside any block {} of code, not just at the start
  - scope is from line of declaration until the end of the code block
  - must be initialized each time block is entered (even inside loops)
- local variable definitions that precede a block
  - method parameters are treated as locals with scope of the method
  - for loop definition can define variables. these are locals
  - scope is the entire block

### 56.1 Shadowing

- **shadowing**: local can have same name as class field or instance field, in which case it is an independent variable.
  - use **ClassName.name** to access shadowed class field
  - use **this.name** to access shadowed instance field
  - local variables cannot be shadowed
    - method parameter *cannot* be shadowed by a local variable inside the method

### 56.2 Obscuring

- When a local variable and a class share the same name, the class is said to be **obscured** by the variable.
- There is no way to reference the class name within the scope of the local

```
public class Weird {  
    static int x =23;  
    void meth(int ron) {  
        int Weird = 99;  
        out.println("int Weird = "+Weird);  
        int y = Weird.x; ✗ compile-time error: class name is obscured by local variable  
    }  
}
```

### 56.3 block{}

- local variables *cannot* be shadowed by another local inside a nested block
- block can only reference variables outside the block if they were declared above the block
- scope of local variable inside a block is from line of declaration to the end of the block. so they can be reused outside

```
int w = 12; ✗ compile-time error: this would clash with w defined in nested block  
  
int y = 0;
```

```
{
    int w = y + 2;
    // cannot reference z in here
}

int z = 3;

{
    int w = y + z + 25; // new w variable!
}

int w = 12; // new w okay here because scope starts here
```

## 56.4 switch body

```
switch (x) {

    // NO CODE ALLOWED HERE

    case 0:
        int w = 13;

        out.println("    w = "+w);
        out.println("    case 0");
        break;
        out.println("    w = "+w);

        // LAST LINE OF w's scope

    case 1:
        out.println("    case 1");
        break;

    default:
        out.println("    default");
        break;

}
```

## 56.5 For loops

```
for(int w = 1; w <= 2; w++) {
    int r = -1; // must initialize each time through loop
    r++;
    out.println("r = "+r);
    out.println("w = "+w);
}
```

### OUTPUT

```
r = 0
w = 1
r = 0
```

```
w = 2
```

## 56.6 Declaration lists

Variables of the same type can be declared and initialized in a list.

```
int a, b = 2, c = 3;  
a = 1;
```

```
Object a = new Object(), b;  
b = new Object();
```

## 57 Method Parameters and Arguments

- **formal parameters** for a method are declared in a list
- each parameter serves as a local variable whose scope is the body of the function
- the value of these local variables is initialized from the **arguments** given at **invocation**

```
void doSomething(int ron, String s, Double d ) {  
}
```

- when the method is **invoked**, a variable or expression must be provided for each parameter.
  - These specified values are called **arguments**

```
String str = "Hello";  
Double h = 3.14;  
obj.doSomething(23, str+" there!", (2*h+0.1)/23.2);
```

- unlike locals, the parameters each have type
- unlike locals, they cannot be initialized
- when invoking the method, every parameter must be specified
- use the builder pattern (ie pass a class instance that contains all of the parameters with defaults values)
- the keyword final can be used, in which case the variable cannot be modified inside the body of the method
- since JAVA SE 8, a **receiver parameter** can be used
  - This means that you can explicitly list **this** as the first parameter
  - You ignore the receiver parameter when you invoke
  - The only use is for annotations

```
class MyClass {  
    int y = 1;  
    int getY(MyClass this) {  
        return y;  
    }  
}
```

```
MyClass ob = new MyClass();  
out.println( ob.getY() );    // passing ob or any other object is an error
```



## 58 varargs

- Java allows a variable number of parameters, a **vararg**, to be specified for a method.
- Once captured the parameters are given to the code as an array

```
void printAll(int ... xargs) {  
    for(int x: xargs){  
        out.println(x);  
    }  
}
```

- only one **vararg** is allowed per method
- **vararg** must be the last parameter in the list
- **vararg** can be of any type including primitives and classes
- any number of arguments can be provided: 0,1,2...
- you **can** immediately precede the **vararg** with a parameter of the same type
  - preceded with one parameter of the same type effectively allows 1 or more arguments of that type:

```
void printAll(int j, int ... xargs) {  
}
```

- Using a vararg as a generic type parameter can cause heap pollution

## 59 Uses of **final** Keyword

### 59.1 class fields

- A **final** class field must be **initialized at declaration**.
- This is effectively a constant.
- Inherited **final** class fields can be hidden by descendents.

### 59.2 instance fields

- An **final** instance field must be initialized
  - at declaration
- OR
  - in every constructor
- OR
  - in a single block (before or after)
- This is effectively a constant for the instance.
- Inherited **final** instance fields can be hidden by descendents.

### 59.3 method parameters

- **final** denotes that the parameter variable cannot be change within the method

### 59.4 local variables

- **final** denotes that the value of the variable cannot be changed

### 59.5 **final** primitives

For a **final** primitive, the value cannot be changed, so it is a constant for the life of its scope.

### 59.6 **final** objects

- When an object variable is marked **final**, its value (ie reference) cannot be changed.
- However, the object's fields **can be changed**!
- The same rule holds for arrays and their elements.

#### Example

```
class MyClass {  
    int y = 1;  
  
    int getY() {  
        return y;  
    }  
  
    void setY(int y) {  
        this.y = y;  
    }  
}
```

#### Usage

```
final MyClass fob = new MyClass();  
out.println(fob.getY());  
fob.setY(99);  
out.println(fob.getY());
```

### Output

```
1  
99
```

## 59.7 Immutable Objects

An **immutable object** is an object whose state (ie fields) cannot be modified.

- At compile, to make an object that cannot have any of its fields modified requires making all the fields defined as **final**.
- At runtime, to make an object immutable is not supported in Java.
  - It requires extra code for each field.
  - as well as a method to set mutable / immutable
  - need to decided if attempt to change are ignored silently or throw an exception
  - typically in Java, the object is cloned instead so that the original object cannot be modified.

## 59.8 final Classes

A class that is declared **final** cannot be subclassed.

## 59.9 final instance methods

An instance method that is declared **final** cannot be redefined (ie overridden) by a descendent class.

## 59.10 final class methods

An class method that is declared **final** cannot be redefined (ie hidden) by a descendent class.

## 60 Uses of **abstract** Keyword

### 60.1 **abstract** class

A class declared as **abstract** cannot be instantiated

### 60.2 **abstract** instance methods

- A method declared as **abstract** is just a stub without a definition.
  - it declares the return type and signature  
`abstract String xyz(int y);`
- A class with one or more **abstract** methods must be declared as **abstract**
- class methods **cannot** be **abstract**

## 61 Uses of **default** keyword

### 61.1 **default** interface method

- To define a method inside an interface, you need to precede the method with the **default** keyword
- otherwise the method must be **abstract** (ie a stub).

## 62 keyword **native**

- The keyword **native** marks a method, that it will be implemented in another language, not in Java.
- It works together with JNI (Java Native Interface).

Native methods were used in the past to write performance critical sections but with Java getting faster this is now less common.

- Native methods are currently needed when
  - You need to call a library from Java that is written in other language.
  - You need to access system or hardware resources that are only reachable from the other language (typically C). Actually, many system functions that interact with real computer (disk and network IO, for instance) can only do this because they call native code.

See Also [Java Native Interface Specification](#)

Minimal example to make things clearer

### Main.java

```
public class Main {
    public native int square(int i);
    public static void main(String[] args) {
        System.loadLibrary("Main");
        System.out.println(new Main().square(2));
    }
}
```

### Main.c

```
#include <jni.h>
#include "Main.h"

JNIEXPORT jint JNICALL Java_Main_square(
    JNIEnv *env, jobject obj, jint i) {
    return i * i;
}
```

### Compile and run

```
sudo apt-get install build-essential openjdk-7-jdk
export JAVA_HOME='/usr/lib/jvm/java-7-openjdk-amd64'
javac Main.java
javah -jni Main
gcc -shared -fpic -o libMain.so -I${JAVA_HOME}/include \
    -I${JAVA_HOME}/include/linux Main.c
java -Djava.library.path=. Main
```

### Output

```
4
```

- Tested on Ubuntu 14.04 AMD64. Also worked with Oracle JDK 1.8.0\_45.
- [Example on GitHub](#) for you to play with.
- Underscores in Java package / file names must be escaped with `_1` in the C function name as mentioned at: [Invoking JNI functions in Android package name containing underscore](#)

### Interpretation

It allows you to:

- call a compiled dynamically loaded library (here written in C) with arbitrary assembly code from Java
- and get results back into Java

This could be used to:

- write faster code on a critical section with better CPU assembly instructions (not CPU portable)
- make direct system calls (not OS portable)

with the tradeoff of lower portability.

It is also possible for you to call Java from C, but you must first create a JVM in C:

[How to call Java functions from C++?](#)

### Android NDK

The concept is exact the same in this context, except that you have to use Android boilerplate to set it up.

The official NDK repository contains "canonical" examples such as the hello-jni app:

- <https://github.com/googlesamples/android-ndk/blob/4df5a2705e471a0818c6b2dbc26b8e315d89d307/hello-jni/app/src/main/java/com/example/hellojni/HelloJni.java#L39>
- <https://github.com/googlesamples/android-ndk/blob/4df5a2705e471a0818c6b2dbc26b8e315d89d307/hello-jni/app/src/main/cpp/hello-jni.c#L27>

In you unzip an .apk with NDK on Android O, you can see the pre-compiled .so that corresponds to the native code under lib/arm64-v8a/libnative-lib.so.

TODO confirm: furthermore, file /data/app/com.android.appname-\*/oat/arm64/base.odex, says it is a shared library, which I think is the AOT precompiled .dex corresponding to the Java files in ART, see also:

[What are ODEX files in Android?](#)

So maybe the Java is actually also run via a native interface?

### Example in the OpenJDK 8

Let's find where `Object#clone` is defined in `jdk8u60-b27`.

We will conclude that it is implemented with a **native** call.

First we find:

```
find . -name Object.java
```

which leads us to [jdk/src/share/classes/java/lang/Object.java#1212](#):

```
protected native Object clone() throws CloneNotSupportedException;
```

Now comes the hard part, finding where `clone` is amidst all the indirection. The query that helped me was:

```
find . -iname object.c
```

which would find either **C or C++ files** that might implement `Object`'s native methods. It leads us to [jdk/share/native/java/lang/Object.c#147](#):

```
static JNINativeMethod methods[] = {
    ...
    {"clone",      "()Ljava/lang/Object;",   (void *)&JVM_Clone},
};

JNIEXPORT void JNICALL
Java_java_lang_Object_registerNatives(JNIEnv *env, jclass cls)
{
    (*env)->RegisterNatives(env, cls,
                           methods, sizeof(methods)/sizeof(methods[0]));
}
```

which leads us to the `JVM_Clone` symbol:

```
grep -R JVM_Clone
```

which leads us to [hotspot/src/share/vm/prims/jvm.cpp#1580](#):

```
JVM_ENTRY(jobject, JVM_Clone(JNIEnv* env, jobject handle))
    JVMWrapper("JVM_Clone");
```

After expanding a bunch of macros, we come to the conclusion that this is the definition point.

## 63 keyword `transient`

Before understanding the `transient` keyword, one has to understand the concept of serialization. If the reader knows about serialization, please skip the first point.

### 63.1 What is serialization?

**Serialization** is the process of making the object's state persistent. That means the state of the object is converted into a stream of bytes to be used for persisting (e.g. storing bytes in a file) or transferring (e.g. sending bytes across a network). In the same way, we can use the deserialization to bring back the object's state from bytes. This is one of the important concepts in Java programming because serialization is mostly used in networking programming. The objects that need to be transmitted through the network have to be converted into bytes. For that purpose, every class or interface must implement the [Serializable](#) interface. It is a marker interface without any methods.

### 63.2 Now what is the `transient` keyword and its purpose?

By default, all of object's variables get converted into a persistent state. In some cases, you may want to avoid persisting some variables because you don't have the need to persist those variables. So you can declare those variables as `transient`. If the variable is declared as `transient`, then it will not be persisted. That is the main purpose of the `transient` keyword.

I want to explain the above two points with the following example:

```
package javabeat.samples;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class NameStore implements Serializable{
    private String firstName;
    private transient String middleName;
    private String lastName;

    public NameStore (String fName, String mName, String lName){
        this.firstName = fName;
        this.middleName = mName;
        this.lastName = lName;
    }

    public String toString(){
        StringBuffer sb = new StringBuffer(40);
        sb.append("First Name : ");
        sb.append(this.firstName);
        sb.append("Middle Name : ");
        sb.append(this.middleName);
        sb.append("Last Name : ");
        sb.append(this.lastName);
        return sb.toString();
    }
}

public class TransientExample{
    public static void main(String args[]) throws Exception {
        NameStore nameStore = new NameStore("Steve", "Middle", "Jobs");
        ObjectOutputStream o = new ObjectOutputStream(new FileOutputStream("nameStore"));
        // writing to object
        o.writeObject(nameStore);
        o.close();

        // reading from object
    }
}
```



```
ObjectInputStream in = new ObjectInputStream(new FileInputStream("nameStore"));
NameStore nameStore1 = (NameStore)in.readObject();
System.out.println(nameStore1);
    }
}
```

And the output will be the following:

```
First Name : Steve
Middle Name : null
Last Name  : Jobs
```

*Middle Name* is declared as **transient**, so it will not be stored in the persistent storage.

## 64 keyword `volatile`

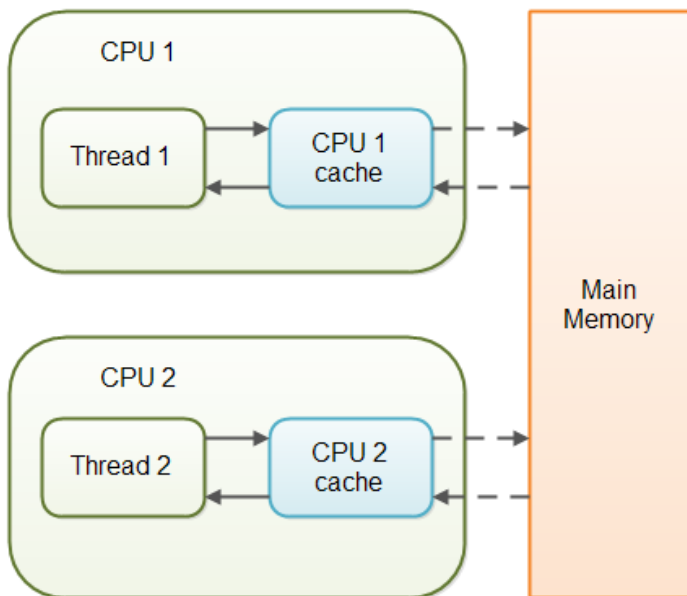
The Java `volatile` keyword is used for multi-threaded applications.

The Java `volatile` keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a `volatile` variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a `volatile` variable will be written to main memory, and not just to the CPU cache.

### 64.1 Variable Visibility Problems

The Java `volatile` keyword guarantees visibility of changes to variables across threads. This may sound a bit abstract, so let me elaborate.

In a multithreaded application where the threads operate on non-volatile variables, each thread may copy variables from main memory into a CPU cache while working on them, for performance reasons. If your computer contains more than one CPU, each thread may run on a different CPU. That means, that each thread may copy the variables into the CPU cache of different CPUs. This is illustrated here:



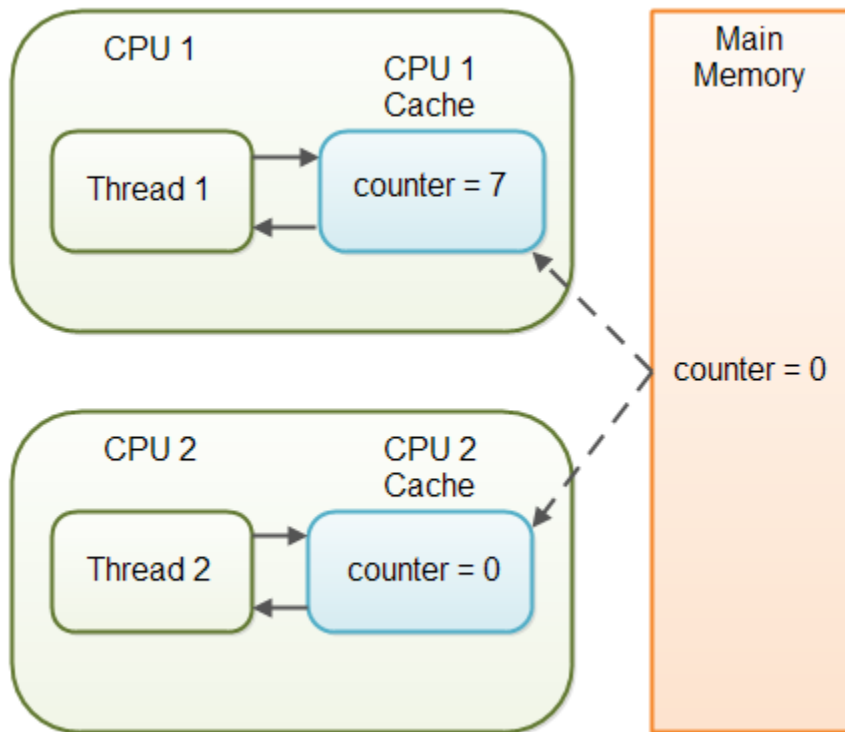
With non-volatile variables there are no guarantees about when the Java Virtual Machine (JVM) reads data from main memory into CPU caches, or writes data from CPU caches to main memory. This can cause several problems which I will explain in the following sections.

Imagine a situation in which two or more threads have access to a shared object which contains a counter variable declared like this:

```
public class SharedObject {  
    public int counter = 0;  
}
```

Imagine too, that only Thread 1 increments the `counter` variable, but both Thread 1 and Thread 2 may read the `counter` variable from time to time.

If the `counter` variable is not declared `volatile` there is no guarantee about when the value of the `counter` variable is written from the CPU cache back to main memory. This means, that the `counter` variable value in the CPU cache may not be the same as in main memory. This situation is illustrated here:



The problem with threads not seeing the latest value of a variable because it has not yet been written back to main memory by another thread, is called a "visibility" problem. The updates of one thread are not visible to other threads.

## 64.2 The Java `volatile` Visibility Guarantee

The Java `volatile` keyword is intended to address variable visibility problems. By declaring the `counter` variable `volatile` all writes to the `counter` variable will be written back to main memory immediately. Also, all reads of the `counter` variable will be read directly from main memory.

Here is how the `volatile` declaration of the `counter` variable looks:

```
public class SharedObject {
    public volatile int counter = 0;
}
```

Declaring a variable `volatile` thus *guarantees the visibility* for other threads of writes to that variable.

In the scenario given above, where one thread (T1) modifies the `counter`, and another thread (T2) reads the `counter` (but never modifies it), declaring the `counter` variable `volatile` is enough to guarantee visibility for T2 of writes to the `counter` variable.

If, however, both T1 and T2 were incrementing the `counter` variable, then declaring the `counter` variable `volatile` would not have been enough. More on that later.

## 64.3 Full `volatile` Visibility Guarantee

Actually, the visibility guarantee of Java `volatile` goes beyond the `volatile` variable itself. The visibility guarantee is as follows:

If Thread A writes to a `volatile` variable and Thread B subsequently reads the same `volatile` variable, then all variables visible to Thread A before writing the `volatile` variable, will also be visible to Thread B after it has read the `volatile` variable.

If Thread A reads a `volatile` variable, then all all variables visible to Thread A when reading the `volatile` variable will also be re-read from main memory.

Let me illustrate that with a code example:

```
public class MyClass {
    private int years;
    private int months;
    private volatile int days;

    public void update(int years, int months, int days) {
        this.years = years;
        this.months = months;
        this.days = days;
    }
}
```

The `update()` method writes three variables, of which only `days` is **volatile**.

The full **volatile** visibility guarantee means, that when a value is written to `days`, then all variables visible to the thread are also written to main memory. That means, that when a value is written to `days`, the values of `years` and `months` are also written to main memory.

When reading the values of `years`, `months` and `days` you could do it like this:

```
public class MyClass {
    private int years;
    private int months;
    private volatile int days;

    public int totalDays() {
        int total = this.days;
        total += months * 30;
        total += years * 365;
        return total;
    }

    public void update(int years, int months, int days) {
        this.years = years;
        this.months = months;
        this.days = days;
    }
}
```

Notice the `totalDays()` method starts by reading the value of `days` into the `total` variable. When reading the value of `days`, the values of `months` and `years` are also read into main memory. Therefore you are guaranteed to see the latest values of `days`, `months` and `years` with the above read sequence.

## 65 Arrays

- An array is not an instance of any class listed in the class tree, however each array is an object and inherits directly from `java.util.Object`

```
(new int[1]) instanceof Object // -> evaluates to true
```

- Arrays of primitives cannot be cast as `Object[]`

```
(new int[1]) instanceof Object[] // -> compile error
```

- Arrays of Objects cannot be cast as `Object[]`

```
(new String[1]) instanceof Object[] // evaluates to true
```

- Arrays inherit all the members of `java.lang.Object`
- Arrays override the method `clone()` inherited from `Object`.
- Arrays implement the field `length`, which contains the number of components of the array. `length` may be positive or zero. It is `public` and `final`.
- Arrays implement the interfaces `Cloneable` and `java.io.Serializable`.
- Arrays are supported by `Class<T>`. You can retrieve the `Class<T>` instance from an array instance

```
(new int[2]).getClass()
```

or from an array type

```
int[].class
```

- A unique reflection class instance (ie an instance of `java.lang.Class<T>`) is created for each different array type in your code. Examples

```
int[].class.getCanonicalName() // -> "int[]"
String[].class.getCanonicalName() // -> "java.lang.String[]"
```

- In effect, each array of a different data type becomes a unique class that inherits from `Object`. That is `int[]` and `String[]` in the code snippet above each effectively define separate classes.
- The class `java.util.Arrays` is a helper class, and arrays are not instances of this class.

```
(new int[1]) instanceof java.util.Arrays // -> compile error
```

- The class `java.lang.reflect.Array` is a helper class, and arrays are not instances of this class.

```
(new int[1]) instanceof java.lang.reflect.Array // -> compile error
```

To repeat: Arrays are objects but are not instances of any class in the class tree.

### SPECIFICATION REFERENCES

From the Java specification [Section 4.3.1 Objects](#)

*An object is a class instance or an array.*

A class instance is explicitly created by a class instance creation expression.

An array is explicitly created by an array creation expression.

From [java.util.Arrays](#)

- This class contains various methods for manipulating arrays (such as sorting and searching)

From [java.lang.reflect.Array](#)

- The Array class provides static methods to dynamically create and access Java arrays.

From [Section 10.1 Objects](#)

The direct superclass of an array type is `Object`.

Every array type implements the interfaces `Cloneable` and `java.io.Serializable`.

From [Section 10.7 Array Members](#)

The members of an array type are all of the following:

The public final field `length`, which contains the number of components of the array. `length` may be positive or zero.

The public method `clone`, which overrides the method of the same name in class `Object` and throws no checked exceptions. The return type of the `clone` method of an array type `T[]` is `T[]`.

A clone of a multidimensional array is shallow, which is to say that it creates only a single new array. Subarrays are shared.

All the members inherited from class `Object`; the only method of `Object` that is not inherited is its `clone` method.

## 65.1 Array elements (aka components)

- All of the elements of an array must be of the same type (either of the same primitive type or of the same class).
- However, you can create an array of class `Object` and then the elements can be any subclass, including `Arrays` and wrapped primitives (but not bare primitives).
- 

## 65.2 Array variable Declaration, instantiation, Initialization

- **declaration:**

```
int[] intvals;
```

- you can put the `[]` brackets after the variable name (discouraged)
- **instance creation.** you must define the length! `length >= 0`

```
(new int[5])
```

- `length` can be any integer expression that evaluates to `>= 0`

```
(new int[i])
(new int[i++])
(new int[i+2*j])
(new int[i=13])
```

- An array can have `length = 0`, in which case it has no elements
- Same as objects. you **must initialize** before using (else compile time error).
- **declaration and initialization via instance creation** combined

```
int[] intvals = new int[5];
```

- Important, this array can now be used. **The array components are initialized to default values:**
  - For references (anything that holds an object, including subarrays) that is `null`.
  - For `int`/`short`/`byte`/`long` that is a `0`.
  - For `float`/`double` that is a `0.0`
  - For `booleans` that is a `false`.
  - For `char` that is the null character `'\u0000'` (whose decimal equivalent is 0).

### 65.3 Arrays variables and null

Same as object variables, an array variable can be assigned to `null`. Then it has no elements and no size. No array exists.

### 65.4 An array of Chars is NOT a String

### 65.5 Array "constants"

- can only be used in an initializer **or** creation expression
- **initializer**

```
int[] intvals = {1,2};
```

- **array instance creation**

```
new int[] {1,2}
```

- components can be variables or expressions!

```
int x = 99;  
int[] intvals = {x+1,2};
```

- set using instance creation

```
int[] intvals;  
intvals = new int[] {100,102};
```

### 65.6 Examples of array declaration and initialization

```
int[] myarray = new int[0];  
→ myarray = {}
```

```
int[] myarray = new int[2];  
→ myarray = {0, 0}
```

```
int[] myarray = {1,2};  
→ myarray = {1, 2}
```

```
int x = 99;  
int[] myarray = {x,2};  
→ myarray = {99, 2}
```

```
String[] myarray = new String[1];  
→ myarray = {null}
```

```
String[] myarray = {"hello","there"};  
→ myarray = {hello, there}
```

```
String s = "!";  
String[] myarray = {"hello","there",s};  
→ myarray = {hello, there, !}
```

## 65.7 Arrays of arrays instead of multidimensional arrays, in contrast to C/C++

- Thus each subarray can have the same size or different sizes.
- Each subarray must be of same type
- since they are components of an array, subarrays are initialized to **null**

## 65.8 examples of multiarrays declarations and initialization

```
int[][] myarray = new int[0][0]  
→ myarray = {}
```

```
int[][] myarray = new int[2][3];  
→ myarray = {{0, 0, 0}, {0, 0, 0}}
```

```
int[][] myarray = new int[2][];  
→ myarray = {null, null}  
myarray [0] = new int[] {11,22,33};  
→ myarray = {{11, 22, 33}, null}  
myarray [1] = new int[3];  
→ myarray = {{11, 22, 33}, {0, 0, 0}}
```

```
int x = 1;  
int[][] myarray = {null,{x++},{x++,x++}};  
→ myarray = {null, {1}, {2, 3}}
```

```
Object[][] myarray = {new Integer[] {1,2}, new Double[] {3.4}, new String[]  
{"hello","world"}};  
→ myarray = {{1, 2}, {3.4}, {hello, world}}
```



# 66 Interfaces

An interface is like a contract or an API that classes can implement.

Interfaces allow instance methods to be defined (using default). This allows stateless, compile-time mixins.

## 66.1 Differences from classes

Interfaces are similar to classes with the following differences.

- defined with keyword **interface** instead of **class**
- interfaces do not have constructors and cannot be instantiated
  - **new** cannot be used with an interface
- an interface cannot have a state (can only have constant fields)
- classes **implement** interfaces instead of **extend**
- classes can implement one or more interfaces (given as a list separated by commas)
- classes always inherit from a parent class, even if an interface(s) is implemented
- an interface can be used as a type anywhere that a class type can be used.
  - an interface can be used as a variable type
  - an interface can be used as an argument type
  - an interface can be used on the right hand side of the **instanceof** operator
  - an instance of a class that implements an interface can be **cast** as that interface
- an interface can **extend** another interface to create a hierarchy of interfaces

### 66.1.1 Fields

- can only contain constant fields (ie must be **final static**)
  - in other words, an interface cannot have a state
  - **final** and **static** are implicit for fields but can be given explicitly if preferred
- all fields can be re-defined in implementing classes or a descendent interface (uses hiding)

### 66.1.2 Methods

- each method must be preceded by zero or one (and no greater than one) of the following keywords: **abstract**, **default**, **static**
- methods defined without **abstract**, **default**, **static** are implicitly preceded with **abstract**
- **abstract** and **default** refer to instance methods. static methods cannot have either keyword.
- **abstract** methods of an interface must be defined in any concrete class that implements it
- to define an instance method that is not abstract, you must use the keyword **default**
- to define a class method you must use **static**
- class methods cannot be **abstract**

### 66.1.3 this and super

- **this** cannot be used in an interface
- **super** cannot be used in an interface

### 66.1.4 No root interface but...

- There is no root interface
  - Unlike classes (which inherit from **Object** if no parent is specified) an interface without a specified parent has no parent
- However, any **object instance** (of any class) whose type is an interface *does* have access to all the members of **Object**
  - The members of **Object** are always defined for any object instance regardless of its type.

- The members of **Object** are not accessible from within the interface definition

### 66.1.5 Class and interface members

- an interface can be defined as a member of an interface
- an interface can be defined as a member of a class
- a class can be defined as a member of an interface
- and of course, a class can be defined as a member of a class

### 66.1.6 lambda interfaces

**TODO**

## 67 Enumerations

An enumeration is a class definition, but it has special built-in syntax and features not available in other classes.

### 67.1 Defining Enum classes

- All enums implicitly extend `java.lang.Enum`
- You cannot explicitly use this class.
- To define an enum class, you must use the `enum` keyword instead of `class`.

### 67.2 Instances are the enum values

- You cannot create instances of an enum class at runtime
- Instead, the instances are created as part of a special initialization statement inside the class
- the instances themselves are the enum values
  - in other words the enum values are the predefined instances of the class
- Because the instances/values are defined inside the class, outside the class they need to be referenced preceded with the enum class name.
- You can always use `==` to compare enums

### 67.3 Simple Example

#### Definition

```
enum Fruit {  
    // instance definitions (aka the "values" of the enum)  
    // new keyword is not used  
    APPLE(), BANANA(), ORANGE();  
  
    // constructor  
    Fruit() {  
    }  
}
```

#### Usage

```
static void printMyOpinion(Fruit f) {  
  
    // even though objects, the enum values can be used inside a switch  
    // compiler infers Fruit type from the variable f  
    // only enum values are allowed as cases  
    switch (f) {  
        case APPLE:  
            out.println("I like Apples");  
            break;  
        case BANANA:  
            out.println("I hate Bananas");  
            break;  
        case ORANGE:  
            out.println("I love Oranges");  
            break;  
    }  
}
```

```
public static void main(String[] args) {  
    Fruit f = Fruit.APPLE; // Fruit. required here  
    printMyOpinion(f);  
}
```

## Output

```
I like Apples
```

### 67.3.1 Shorthand

- The default constructor can be omitted and the compiler will create one for you.
- The parentheses in the invocations of the default constructor can be omitted

From these two rules, we can shorten the definition as follows.

## Equivalent Definition

```
enum Fruit {  
    APPLE, BANANA, ORANGE;  
}
```

## 67.4 Inherited methods

### 67.4.1 static method: `values()`

This method returns an array containing all the values of the enum class in ordinal order.

```
for (Fruit f : Fruit.values()) {  
    out.println(f);  
}
```

### 67.4.2 static method: `valueOf()`

```
Enum.valueOf(Fruit.class, "ORANGE") → Fruit.ORANGE
```

### 67.4.3 methods: `name()`, `toString()`

```
APPLE.name() → "APPLE"  
APPLE.toString() → "APPLE"
```

### 67.4.4 method: `ordinal()`

Returns the ordinal of this enumeration constant (its position in its enum declaration, where the initial constant is assigned an ordinal of zero).

```
Fruit.ORANGE.ordinal() → 2
```

#### 67.4.5 method: `getDeclaringClass()`, `getClass()`

```
Fruit.ORANGE.getDeclaringClass() → Class<Fruit>  
Fruit.ORANGE.getClass() → Class<Fruit>
```

### 67.5 Class functionality

All normal class functionality can be used inside an **enum** class.

- can implement interfaces
- can define fields and methods
- can define different constructors
- can set enum variables to any enum value
- can set enum variables to null

### 67.6 Example

#### Definition

```
enum Fruit {  
    // instance definitions  
    APPLE(0.50, "red"), BANANA(0.80, "yellow"), ORANGE(1.00, "orange");  
  
    // constructor  
    Fruit(Double price, String color) {  
        this.price = price;  
        this.color = color;  
    }  
  
    private Double price;  
    private String color;  
  
    Double getPrice() {  
        return this.price;  
    }  
    String getColor() {  
        return this.color;  
    }  
  
    Fruit setPrice(Double price) {  
        this.price = price;  
        return this;  
    }  
    Fruit setColor(String color) {  
        this.color = color;  
        return this;  
    }  
}
```

#### Usage

```
Fruit.APPLE.getPrice()    → 0.5  
Fruit.APPLE.getColor()  → "red"
```

## 68 Generics

Generics is the term used to described classes, interfaces, and methods that use a type parameter.

### 68.1 Parametized Class

Java allows the use of **formal type parameters** so that one or more of the data types used by a class can be specified at instantiation.

- The type parameter is specified by the following syntax

```
class Box <T>
```

- At instantiation the type is specified by a concrete type argument

```
new Box<Integer>();
```

- The Type parameter must be a reference type (class, enum, interface, or array).

### 68.2 Example

```
class Box <T> {  
    Box() {  
        this(null);  
    }  
    Box(T value) {  
        this.value = value;  
    }  
    private T value;  
    void set(T value) {  
        this.value = value;  
    }  
    T get() {  
        return this.value;  
    }  
    public String toString() {  
        String s;  
        if (this.value == null) {  
            s = "[null]";  
        } else {  
            s = (this.value).toString();  
        }  
        return "Box containing{" + s + "}";  
    }  
}  
class SubBox <T> extends Box<T>{}
```

#### Usage

```
// fully explicit declaration/initialization  
Box<Integer> box1 = new Box<Integer>(100);  
out.println("box1/2 = " + (box1.get()/2) );
```

```

// diamond (<>) shorthand
Box<Integer> box2 = new Box<>(1);
out.println("box2 = " + box2 );

// var shorthand
var box3 = new Box<Integer>(1+1);
out.println("box3 = " + box3 );

// Box of a Box<Integer>
Box<Box<Integer>> boxobox = new Box<Box<Integer>>( new Box<Integer>(99) );
out.println("boxobox = " + boxobox );

// primitives are not allowed, but arrays of primitives are!
double[] darray = new double[] {5.5, 6., 6.5};
Box<double[]> box4 = new Box<>(darray);
out.print("box4 = { ");
for (int i = 0; i < box4.get().length; i++) {
    if (i > 0) out.print(", ");
    out.print(box4.get()[i] );
}
out.println(" }");

```

## Output

```

box1/2 = 50
box2 = Box containing{1}
box3 = Box containing{2}
boxobox = Box containing{Box containing{99}}
box4 contents = { 5.5, 6.0, 6.5 }

```

## 68.3 Rules for Parametized Classes (Interfaces, Enums)

- **T** cannot be a primitive
- The <T> comes immediately followign the class name
- More than one type parameter can be specified using a list

```
class Box <T,V>
```

- Naming convention for type parameters

```

E - Element (used extensively by the Java Collections Framework)
K - Key
N - Number
T - Type (general type)
V - Value
S,U,V etc. - 2nd, 3rd, 4th types

```

- The type argument can be a concrete parameterized class, even the parameterized class itself

```

var box5 = new Box< ArrayList<Integer> >();
var box6 = new Box< Box<Integer> >();

```

- You can bound the type **from above** (**T** ≤ *TopClass*) using the following syntax .



```
class Test <T extends Number> {}
```

- This means that the type supplied as an argument **must be** `Number` or a **subclass** of `Number`

- **EXAMPLE USAGE**

```
var test1 = new Test<Integer>(); // ✓ Integer ≤ Number
var test2 = new Test<Number>(); // ✓ Number ≤ Number
var test3 = new Test<Object>(); ✗ compile-time error: Object is a superclass of Number
```

- You **cannot** bound the type **from below** ( $T \geq \text{BottomClass}$ ).
  - This is in contrast to the wildcard `<?>`
- You can **bound from above** using an **interface**
  - the keyword `extends` is used in this context as well
  - the type can be any subinterface **or** any class that implements this interface or its subinterfaces

```
class Test <T extends Cloneable> {} // java.lang.Cloneable is an interface
```

- **EXAMPLE USAGE**

```
var test1 = new Test<Name>(); // ✓ interface javax.naming.Name is a subinterface to Cloneable
var test2 = new Test<CompositeName>(); // ✓ class javax.naming.CompositeName implements
// Name
var test3 = new Test<Object>(); ✗ compile-time error: Object does not implement Cloneable
```

- You can **bound from above** using a **class and/or multiple interfaces** using the `&` operator
  - This is termed an **intersection type**

```
class Test <T extends Number & Serializable & MyInterface> {}
```

- You **cannot** create **arrays of parameterized types**

```
Integer[] array = new Integer[0]; // ✓
Box<Integer>[] array2 = new Box<Integer>[0]; ✗ compile-time error: no arrays of generics
```

Problems arise when an array holds elements whose type is a concrete parameterized type. Because of type erasure, parameterized types do not have exact runtime type information. As a consequence, the array store check does not work because it uses the dynamic type information regarding the array's (non-exact) component type for the array store check. Example (of array store check in case of parameterized component type):

```
Pair<Integer,Integer>[] intPairArr = new Pair<Integer,Integer>[10]; // illegal
Object[] objArr = intPairArr;
objArr[0] = new Pair<String,String>("", ""); // should fail, but would succeed
```

If arrays of concrete parameterized types were allowed, then a reference variable of type `Object[]` could refer to a `Pair<Integer,Integer>[]`, as shown in the example. At runtime an array store check must be performed when an array element is added to the array. Since we are trying to add a `Pair<String,String>` to a `Pair<Integer,Integer>[]` we would expect that the type check fails. However, the JVM cannot detect any type mismatch here: at runtime, after type erasure, `objArr` would have the dynamic type `Pair[]` and the element to be stored has the matching dynamic type `Pair`. Hence the store check succeeds, although it should not. If it were permitted to declare arrays that holds elements whose type is a concrete parameterized type we would end up in an unacceptable situation. The array in our example would contain different types of pairs instead of pairs of the same type. This is in contradiction to the expectation that arrays hold elements of the same type (or subtypes thereof). This undesired situation would most likely lead to program failure some time later, perhaps when a method is invoked on the array elements.

Example (of subsequent failure):

```
Pair<Integer,Integer>[] intPairArr = new Pair<Integer,Integer>[10]; // illegal
Object[] objArr = intPairArr;
objArr[0] = new Pair<String,String>("", ""); // should fail, but would succeed
Integer i = intPairArr[0].getFirst(); // fails at runtime with ClassCastException
```

The method `getFirst` is applied to the first element of the array and it returns a `String` instead of an `Integer` because the first element in the array `intPairArr` is a pair of strings, and not a pair of integers as one would expect. The innocent-looking assignment to the `Integer` variable `i` will fail with a `ClassCastException`, although no cast expression is present in the source code. Such an unexpected `ClassCastException` is considered a violation of type-safety.

In order to prevent programs that are not type-safe all arrays holding elements whose type is a concrete parameterized type are illegal. For the same reason, arrays holding elements whose type is a wildcard parameterized type are banned, too. Only arrays with an unbounded wildcard parameterized type as the component type are permitted. More generally, reifiable types are permitted as component type of arrays, while arrays with a non-reifiable component type are illegal.

- Generic classes **cannot** implement `Throwable`, ie generic Exceptions are not allowed. (The two children of `Throwable` are `Exception` and `Error`).

## 68.4 Subclassing a parameterized class

### 68.4.1 Definition of subclass for parameterized types

- Let **B** be a subclass of **A**, ie **B** < **A**. Let both be parameterized types. The syntax for this relationship is

#### Example

```
class A <T> {}
class B <T> extends A <T> {}
```

#### Usage (Concrete types)

```
A<Integer> a = new A<>();
B<Integer> b = new B<>();
```

### 68.4.2 Definition of subclass for concrete parameterized types

When dealing with **concrete** versions of these classes,  
**B<Class2> is only a subclass of A<Class1> if Class2==Class1**

- `Integer` is a subclass of `Number`
- **B** < **T** is a subclass of **A** < **T**

- o `B<Integer>` is a subclass `A<Integer>`
- o `A<Integer>` is **not** a subclass of `A<Number>`
- o `B<Integer>` is **not** a subclass of `A<Number>`
- o `B<Integer>` is a subclass `A<?>`
- o `B<Integer>` is a subclass `B<?>`

We cannot cast `A<Integer>` as `A<Number>` !

We cannot cast `B<Integer>` as `A<Number>` !

## CODE EXAMPLES

```
B<?> b2 = b;           // ✓ can cast as wildcard type B<?>
A<?> b3 = (A<?>)b;      // ✓ can cast as wildcard type of superclass A<?>
A<Integer> b4 = (A<Integer>) b; // ✓ can cast as superclass with same argument type
B<Number> b5 = (B<Number>)b; → ✗ compile-time error: even though Integer is a subtype
                                     of Number
A<Number> b6 = (A<Number>)b; → ✗ compile-time error:
```

### 68.4.3 Limitations of `instanceof`

- **Caveat:** `instanceof` only works with the raw or wildcard type, not concrete types

```
(a instanceof A)           → true    // raw type--deprecated
(a instanceof A<?>)        → true    // wildcard--preferred
(a instanceof A<Integer>)  → ✗ compile-time error: concrete types not allowed
(b instanceof A<?>)        → true
```

- o this limitation makes it impossible in the general case to determine if a concrete parameterized instance can be cast as another concrete parameterized class
- o Of course, if the cast can be determined to be improper **at compile-time**, then an error will be generated
- o This does leave open the possibility of generating run-time cast exceptions with no mechanism to check

### 68.4.4 Type Erasure and its consequence

- The compiler knows about the concrete parameterized types and looks for any cast errors that it can find.
- However, after compilation **type erasure** occurs: `T` is replaced with `Object`
- Thus, the concrete parameterized type is unknown at run-time!
- A consequence of this is that you cannot instantiate objects of type `T`! `new T()` does not compile.
- The second consequence is (as mentioned above) the `instanceof` operator cannot check against concrete type
- Also `T.class` does not compile.
- `x instanceof T` does not compile

### 68.4.5 Getting around type erasure

The following is an example of how to store the run-time type and how to create a new instance.

#### EXAMPLE

```
public class GenericClass<T> {
    private final Class<T> classOfT;
```

```

public GenericClass(Class<T> classOfT) {
    this.classOfT = classOfT;
}

public Class<T> getClassOfT() {
    return this.classOfT;
}

public boolean isTypeSame(GenericClass<?> obj) {
    return this.getClassOfT() == obj.getClassOfT();
}

public void printIsTypeSame(GenericClass<?> obj) {
    boolean flag = this.getClassOfT() == obj.getClassOfT();
    if (flag) {
        out.println("Objects have same Element type");
    } else {
        out.println("Objects do NOT have same Element type");
    }
}
}

```

## USAGE

Assume the simple class definition `class Test {}`

```

GenericClass<String> a = new GenericClass<>(String.class);
out.println("Concrete type of a is GenericClass<"+ a.getClassOfT().getSimpleName() + ">");

var b = new GenericClass<>(Integer.class);
out.println("Concrete type of b is GenericClass<"+ b.getClassOfT().getSimpleName() + ">");

out.print("Compare a to a: ");
a.printIsTypeSame(a);
out.print("Compare a to b: ");
a.printIsTypeSame(b);

try {
    var obj = new GenericClass<>(Test.class);
    // can call newInstance() directly from Class<T> in older version of Java
    Constructor<?> c = obj.getClassOfT().getDeclaredConstructor();
    var newObj = c.newInstance();
    out.println("Type of newObj is "+ newObj.getClass().getSimpleName());
} catch (Exception e) {
}

```

## OUTPUT

```

Concrete type of a is GenericClass<String>
Concrete type of b is GenericClass<Integer>
Compare a to a: Objects have same Element type
Compare a to b: Objects do NOT have same Element type
Type of newObj is Test

```

## 68.5 Wildcard Type Parameters

- The wildcard parameter is the `?` symbol: `GenericClass<?>`.
- This form can be used anywhere a type can be used:
  - variable type
  - method parameter type
  - return type
- You cannot instantiate Wild card types (This is the one difference between the wildcard and the raw type)
- You can bound the type **from above** (`? ≤ TopClass`) using the following syntax.

```
GenericClass<? extends Number> x;
```

- This means that the type supplied as an argument must be `Number` or a subclass of `Number`
  - In addition, you can bound the type **from below** (`? ≥ BottomClass`) using the following syntax.
- ```
GenericClass<? super Number> x;
```
- This means that the type supplied as an argument must be `Number` or a superclass of `Number`
  - Wildcards do **not** support *intersection types*
  - Wildcards **do not support both upper and lower bounds** in the same definition

## 68.6 What is a raw type `MyClass`

You can do anything with a raw type `MyClass`.

It combines all the features of `MyClass<Object>` and `MyClass<?>`

## 68.7 Differences between `MyClass<Object>` and `MyClass<?>`

`Box<Object>` = a `Box` containing an `Object`

`Box<?>` = A `Box` of containing an unknown type

- `Box<Integer>` can hold an `Integer`

```
Box<Integer> box = new Box<Integer>();
Integer m = 10;
box.set(m);
```

- `Box<Object>` can hold anything, including an `Integer`

```
Box<Object> box = new Box<Object>();
Integer m = 8;
box.set(m);
```

- `Box<Object>` cannot be assigned to `Box<Integer>`

```
Box<Object> box = new Box<Integer>(); ❌ compile-time error
```

- `Box<?>` can be assigned to `Box<Object>`, `Box<Number>`, or `Box<Integer>`

```
Box<?> box = new Box<Object>();
box = new Box<Number>();
box = new Box<Integer>();
```

- Box<?> can hold no objects, only **null**!

```
Box<?> box = new Box<Integer>();
Integer m = 8;
box.set(m); ✗ compile-time error
box.set(null); ✓
```

- Box<? **extends** Number> can be assigned to Box<Number> or Box<Integer>

```
Box<? extends Number> box;
box = new Box<Object>(); ✗ compile-time error
box = new Box<Number>();
box = new Box<Integer>();
```

- Box<? **extends** Number> can hold no objects, only **null**!

```
Box<? extends Number> box = new Box<Integer>();
Integer m = 8;
box.set(m); ✗ compile-time error
box.set(null); ✓
```

## Summary of assignments

```
List<Object> list; // can only be assigned to a list of type List<Object>
List<?> list; // can be assigned to List<AnyClass>
List<Number> list; // can only be assigned to a list of type List<Number>
List<? extends Number> list; // can be assigned to List<A>, where A is Number or a subclass
```

## Methods have the same issue

```
public void print(List<Number> list);
```

will actually only take a List which is **exactly** List<Number>. It will not take a list which is declared List<Integer>.

```
public void print(List<? extends Number> list);
```

will actually take a List like List<Number>, or any subclass of Number like List<Integer>.

## Using the Box<T> class defined earlier.

```
// Box<Integers> can hold Integers
Box<Integer> boxInteger = new Box<Integer>(1);
out.println("boxInteger = " + boxInteger );

// Box<Number> can hold Integers
Box<Number> boxNumber = new Box<Number>();
Integer n = 10;
boxNumber.set(n);
out.println("boxNumber = " + boxNumber );

// Box<Number> cannot be assigned to a Box<Integer>
Box<Number> boxInteger2 = new Box<Integer>(1); ✗ compile-time error
```

```
// Box<? extends Number> can be assigned to a Box<Integer>
Box<? extends Number> boxInteger2 = new Box<Integer>(1);

// Box< Box<Number> > can hold Box<Number>
Box< Box<Number> > boxbox;
boxbox = new Box< Box<Number> >(boxNumber);
out.println("boxbox = " + boxbox );

// Box< Box<Number> > cannot hold Box<Integer>
boxbox = new Box<Box<Integer>> (boxInteger); ✗ compile-time error: can't assign to a
  Box of Box<Integer>
boxbox = new Box<Box<Number>> (boxInteger); ✗ compile-time error: can't cast boxInteger

// keep trying: this only works for SubBox<Number>
Box< ? extends Box<Number> > → ✓ can also be assigned to Box< Box<Number> >
                             → ✓ can now be assigned to Box< SubBox<Number> >
                             → ✗ still cannot hold Box<Integer>

// keep trying
Box< Box<? extends Number> > → why?2 ✗ cannot be assigned to Box< Box<Number> >
                             → ✗ still cannot hold Box<Integer>

// success ✓
Box< ? extends Box<? extends Number> > boxbox;
boxbox = new Box< Box<Number> >(boxNumber);
boxbox = new Box< Box<Integer> > (boxInteger);

// success ✓ but no type checking whatsoever
Box<?> boxbox = new Box< Box<Number> >(boxNumber);
boxbox = new Box< Box<Integer> > (boxInteger);
```

## 68.8 Generic methods

- Parameterized types can also be used in methods, independent of the method's class parameterized types (if any).
- The principle is the same as a generic class
- The type parameter is placed **immediately before the return type**

```
public final <U> boolean isEmpty(Collection<U> c) {
    return c.isEmpty();
}
```

<sup>2</sup> This probably doesn't work because when we substitute `? extends Number` with `Integer` it breaks

## 69 Generics Example Zoo

### 69.1 class B <T> extends A <T>

As mentioned earlier, this is how to subclass a generic class.

```
class B <T> extends A <T>{}
```

You can add more parameters as well.

```
class B <T,U> extends A <T>{}
```

### 69.2 Forcing class B to use A<T> but not a generic type T

```
class Box <T> {  
    //see earlier definition  
}  
  
class Worker <T> {  
    Worker(Box<T> box) {  
        this.box = box;  
    }  
  
    private Box<T> box;  
  
    void receive(Box<T> box) {  
        this.box = box;  
    }  
  
    Box<T> give() {  
        Box<T> box = this.box;  
        this.box = null;  
        return box;  
    }  
  
    boolean hasBox() {  
        return (this.box != null);  
    }  
}
```

#### USAGE

```
Worker<String> w = new Worker<String>( new Box<String>("hello") );  
out.println("w = " + w );
```

#### OUTPUT

```
w = Worker holding{Box containing{hello}}
```

### 69.3 class B <T extends A<Integer>>

This gives the same functionality as 69.2 with Worker<Integer>



## CODE

```
class WorkerInteger <T extends Box<Integer>> {
    WorkerInteger(T box) {
        this.box = box;
    }

    private T box;

    void receive(T box) {
        this.box = box;
    }


    T give() {
        T box = this.box;
        this.box = null;
        return box;
    }

    boolean hasBox() {
        return (this.box != null);
    }

    public String toString() {
        String s;
        if (this.box == null) {
            s = "[null]";
        } else {
            s = (this.box).toString();
        }
        return "WorkerInteger holding{" + s + "}";
    }
}
```

## USAGE

```
WorkerInteger<Box<Integer>> w = new WorkerInteger<Box<Integer>>( new Box<Integer>(1) );
out.println("w = " + w );
```

WorkerInteger<Integer> ww; →  can only hold Box<Integer>

## OUTPUT

```
w = WorkerInteger holding{Box containing{1}}
```

## 70 Reference Types Definitions

### 4.3 Reference Types and Values

There are five kinds of *reference types*:

- **class** types ([§8.1](#)),
- **enum** types (which are actually a special kind of class type)
- **interface** types ([§9.1](#))
- **type variables** ([§4.4](#)) e.g. `<T, E>`
  - A **type variable** is an unqualified identifier used as a type in class, interface, method, and constructor bodies.
  - A **type variable** is introduced by the declaration of a **type parameter** of a generic class, interface, method, or constructor ([§8.1.2](#), [§9.1.2](#), [§8.4.4](#), [§8.8.4](#)).
  - A class or interface is **generic** if it declares one or more **type variables** ([§4.4](#)).
  - These **type variables** are known as the **type parameters** of the interface. The type parameter section follows the interface name and is delimited by angle brackets.
- **array types** ([§10.1](#)).

ReferenceType:

[ClassOrInterfaceType](#)  
[TypeVariable](#)  
[ArrayType](#)

ClassOrInterfaceType:

[ClassType](#)  
[InterfaceType](#)

ClassType:

[{Annotation}](#) [TypeIdentifier](#) [[TypeArguments](#)]  
[PackageName](#) . [{Annotation}](#) [TypeIdentifier](#) [[TypeArguments](#)]  
[ClassOrInterfaceType](#) . [{Annotation}](#) [TypeIdentifier](#) [[TypeArguments](#)]

InterfaceType:

[ClassType](#)

TypeVariable:

[{Annotation}](#) [TypeIdentifier](#)

ArrayType:

[PrimitiveType](#) [Dims](#)  
[ClassOrInterfaceType](#) [Dims](#)  
[TypeVariable](#) [Dims](#)

Dims:

[{Annotation}](#) [ ] [{Annotation}](#) [ ] }

The sample code:

```
class Point { int[] metrics; }
interface Move { void move(int deltax, int deltax); }
```

declares a class type `Point`, an interface type `Move`, and uses an array type `int[]` (an array of `int`) to declare the field `metrics` of the class `Point`.

A class or interface type consists of an identifier or a dotted sequence of identifiers, where each identifier is optionally followed by type arguments ([§4.5.1](#)). If type arguments appear anywhere in a class or interface type, it is a parameterized type ([§4.5](#)).

Each identifier in a class or interface type is classified as a package name or a type name ([§6.5.1](#)). Identifiers which are classified as type names may be annotated. If a class or interface type has the form `T.id` (optionally followed by type arguments), then `id` must be the simple name of an accessible member type of `T` ([§6.6](#), [§8.5](#), [§9.5](#)), or a compile-time error occurs. The class or interface type denotes that member type.

## 71 Heap pollution (4.12.2)

It is possible that a variable of a parameterized type will refer to an object that is not of that parameterized type. This situation is known as *heap pollution*.

Heap pollution can only occur if the program performed some operation involving a raw type that would give rise to a compile-time unchecked warning (§4.8, §5.1.6, §5.1.9, §8.4.1, §8.4.8.3, §8.4.8.4, §9.4.1.2, §15.12.4.2), or if the program aliases an array variable of non-reifiable element type through an array variable of a supertype which is either raw or non-generic.

For example, the code:

```
List l = new ArrayList<Number>();
List<String> ls = l; // Unchecked warning
```

gives rise to a compile-time unchecked warning, because it is not possible to ascertain, either at compile time (within the limits of the compile-time type checking rules) or at run time, whether the variable `l` does indeed refer to a `List<String>`.

If the code above is executed, heap pollution arises, as the variable `ls`, declared to be a `List<String>`, refers to a value that is not in fact a `List<String>`.

The problem cannot be identified at run time because type variables are not reified, and thus instances do not carry any information at run time regarding the type arguments used to create them.

In a simple example as given above, it may appear that it should be straightforward to identify the situation at compile time and give an error. However, in the general (and typical) case, the value of the variable `l` may be the result of an invocation of a separately compiled method, or its value may depend upon arbitrary control flow. The code above is therefore very atypical, and indeed very bad style.

Furthermore, the fact that `Object[]` is a supertype of all array types means that unsafe aliasing can occur which leads to heap pollution. For example, the following code compiles because it is statically type-correct:

```
static void m(List<String>... stringLists) {
    Object[] array = stringLists;
    List<Integer> tmpList = Arrays.asList(42);
    array[0] = tmpList;           // (1)
    String s = stringLists[0].get(0); // (2)
}
```

Heap pollution occurs at (1) because a component in the `stringLists` array that should refer to a `List<String>` now refers to a `List<Integer>`. There is no way to detect this pollution in the presence of both a universal supertype (`Object[]`) and a non-reifiable type (the declared type of the formal parameter, `List<String>[]`). No unchecked warning is justified at (1); nevertheless, at run time, a `ClassCastException` will occur at (2).

A compile-time unchecked warning will be given at any invocation of the method above because an invocation is considered by the Java programming language's static type system to create an array whose element type, `List<String>`, is non-reifiable (§15.12.4.2). If and only if the body of the method was type-safe with respect to the variable array parameter, then the programmer could use the `SafeVarargs` annotation to silence warnings at invocations (§9.6.4.7). Since the body of the method as written above causes heap pollution, it would be completely inappropriate to use the annotation to disable warnings for callers.

Finally, note that the `stringLists` array could be aliased through variables of types other than `Object[]`, and heap pollution could still occur. For example, the type of the array variable could be `java.util.Collection[]` - a raw element type - and the body of the method above would compile without warnings or errors and still cause heap pollution. And if the Java SE Platform defined, say, `Sequence` as a non-generic supertype of `List<T>`, then using `Sequence` as the type of array would also cause heap pollution.

The variable will always refer to an object that is an instance of a class that represents the parameterized type.

The value of `ls` in the example above is always an instance of a class that provides a representation of a `List`.

Assignment from an expression of a raw type to a variable of a parameterized type should only be used when combining legacy code which does not make use of parameterized types with more modern code that does.

If no operation that requires a compile-time unchecked warning to be issued takes place, and no unsafe aliasing occurs of array variables with non-reifiable element types, then heap pollution cannot occur. Note that this does not imply that heap

*pollution only occurs if a compile-time unchecked warning actually occurred. It is possible to run a program where some of the binaries were produced by a compiler for an older version of the Java programming language, or from sources that explicitly suppressed unchecked warnings. This practice is unhealthy at best.*

*Conversely, it is possible that despite executing code that could (and perhaps did) give rise to a compile-time unchecked warning, no heap pollution takes place. Indeed, good programming practice requires that the programmer satisfy herself that despite any unchecked warning, the code is correct and heap pollution will not occur.*

## 72 Type erasure

### Type Erasure (Java Tutorial)

Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming. To implement generics, the Java compiler applies type erasure to:

- Replace all type parameters in generic types with their bounds or `Object` if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
- Insert type casts if necessary to preserve type safety.
- Generate bridge methods to preserve polymorphism in extended generic types.

Type erasure ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead.

### Type Erasure (4.6)

Type erasure is a mapping from types (possibly including parameterized types and type variables) to types (that are never parameterized types or type variables). We write  $|T|$  for the erasure of type  $T$ . The erasure mapping is defined as follows:

- The erasure of a parameterized type (§4.5)  $G<T_1, \dots, T_n>$  is  $|G|$ .
- The erasure of a nested type  $T.C$  is  $|T|.C$ .
- The erasure of an array type  $T[]$  is  $|T|[]$ .
- The erasure of a type variable (§4.4) is the erasure of its leftmost bound.
- The erasure of every other type is the type itself.

Type erasure also maps the signature (§8.4.2) of a constructor or method to a signature that has no parameterized types or type variables. The erasure of a constructor or method signature  $s$  is a signature consisting of the same name as  $s$  and the erasures of all the formal parameter types given in  $s$ .

The return type of a method (§8.4.5) and the type parameters of a generic method or constructor (§8.4.4, §8.8.4) also undergo erasure if the method or constructor's signature is erased.

The erasure of the signature of a generic method has no type parameters.

| <i>parameterized type</i>                     | <i>type erasure</i>     |
|-----------------------------------------------|-------------------------|
| <code>List&lt;String&gt;</code>               | <code>List</code>       |
| <code>Map.Entry&lt;String, Long&gt;</code>    | <code>Map.Entry</code>  |
| <code>Pair&lt;Long, Long&gt;[]</code>         | <code>Pair[]</code>     |
| <code>Comparable&lt;? super Number&gt;</code> | <code>Comparable</code> |

**type parameters****type erasure**

|                                       |                |
|---------------------------------------|----------------|
| <T>                                   | Object         |
| <T extends Number>                    | Number         |
| <T extends Comparable<T>>             | Comparable     |
| <T extends Cloneable & Comparable<T>> | Cloneable      |
| <T extends Object & Comparable<T>>    | Object         |
| <S, T extends S>                      | Object, Object |

## Examples

|                          |                                                                                                                                                                                                                                                  |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Source code</b>       | <pre>//Here, T is bounded by Object i.e. java.lang.Object class GFG&lt;T&gt; {     //Here, T will be replaced by default i.e. Object     T obj;      GFG(T o)     {         obj = o;     }     T getob()     {         return obj;     } }</pre> |
| <b>After compilation</b> | <pre>class GFG {     // Here, T will be replaced by default i.e. Object     Object obj;     GFG(Object o)     {         obj=o;     }     Object getob()     {         return obj;     } }</pre>                                                  |

|                          |                                                                                                                                                                                               |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Source code</b>       | <pre>class Geeks&lt;T extends Integer&gt; {      //Here, T will be replaced by Integer     T n;     Geeks(T o)     {         n = o;     }     T getob()     {         return n;     } }</pre> |
| <b>After compilation</b> | <pre>class Geeks {      //Here, T will be replaced by Integer     Integer n;      Geeks(Integer o)     {</pre>                                                                                |

```
        n = 0;  
    }  
    Integer getob()  
    {  
        return n;  
    }  
}
```

## 73 Reifiable Types

Because **some type information is erased during compilation**, not all types are available at run time.

**Types that are completely available at run time** are known as *reifiable types*.

A type is *reifiable* if and only if one of the following holds:

- It is a primitive type (§4.2).
- It is an array type (§10.1) whose element type is reifiable.
- It refers to a non-generic class
- It refers to an interface type declaration.
- It is a raw type (§4.8).
- It is a parameterized type in which all type arguments are unbounded wildcards (?) (§4.5.1).
- It is a nested type where, for each type **T** separated by a **.**, **T** itself is reifiable.

For example, if a generic class **X<T>** has a generic member class **Y<U>**, then:

- the type **X<?>.Y<?>** is reifiable because **X<?>** is reifiable and **Y<?>** is reifiable.
- The type **X<?>.Y<Object>** is not reifiable because **Y<Object>** is not reifiable.

**An intersection type is *not* reifiable.**

### 73.1



## 74 REFLECTION (RUN-TIME INFORMATION)

### 74.1 Run-time information classes

- `Class`
- `Type`
- `Field`
- `Method`
- `Enumeration`
- `Interface`
- `Package`

### 74.2 Classes

.class examples

### 74.3 Class `Class<T>`

```
java.lang.Object  
java.lang.Class<T>
```

T - the type of the class modeled by this `Class` object. For example, the type of `String.class` is `Class<String>`. Use `Class<?>` if the class being modeled is unknown.

Instances of the class `Class` represent classes and interfaces in a running Java application. An enum is a kind of class and an annotation is a kind of interface. Every array also belongs to a class that is reflected as a `Class` object that is shared by all arrays with the same element type and number of dimensions. The primitive Java types (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double`), and the keyword `void` are also represented as `Class` objects.

`Class` has no public constructor. Instead `Class` objects are constructed automatically by the Java Virtual Machine as classes are loaded and by calls to the `defineClass` method in the class loader.

The following example uses a `Class` object to print the class name of an object:

```
void printClassName(Object obj) {  
    System.out.println("The class of " + obj +  
        " is " + obj.getClass().getName());  
}
```

It is also possible to get the `Class` object for a named type (or for `void`) using a class literal. See Section 15.8.2 of *The Java™ Language Specification*. For example:

```
System.out.println("The name of class Foo is: "+Foo.class.getName());
```

### 74.4 Primitives

The `.class` syntax is a more convenient and the preferred way to obtain the `Class` for a primitive type; however there is another way to acquire the `Class`. Each of the primitive types and `void` has a wrapper class in `java.lang` that is used for boxing of primitive types to reference types. Each wrapper class contains a field named `TYPE` which is equal to the `Class` for the primitive type being wrapped.

```
Class c = Double.TYPE;
```

There is a class `java.lang.Double` which is used to wrap the primitive type `double` whenever an `Object` is required. The value of `Double.TYPE` is identical to that of `double.class`.

```
Class c = Void.TYPE;
```

`Void.TYPE` is identical to `void.class`.

```
out.println("\u27F6 Integer.class = "+Integer.class);
out.println("\u27F6 Integer.TYPE = "+Integer.TYPE);
out.println("\u27F6 int.class = "+int.class);
```

```
→ Integer.class = class java.lang.Integer
→ Integer.TYPE = int
→ int.class = int
```

```
int x = 0;
out.println("\u27F6 myclass = "+((Object) x).getClass());
try {
    Class<?> myclass = ((Object) x).getClass();
    out.println("\u27F6 myclass = "+myclass.getField("SIZE").get(myclass));
    out.println("\u27F6 myclass = "+myclass.getField("TYPE").get(myclass));
} catch (Exception e) {
}
out.println("");
```

```
→ myclass = class java.lang.Integer
→ myclass = 32
→ myclass = int
```

## 74.5 Objects

## 74.6 Arrays

## 74.7 Generics (Parametized Types)

type erasure and raw types. wildcards. `<Object>`

## 74.8 Interfaces

# 75 Hacks

## 75.1 println shortcuts

### OPTION 1

```
PrintStream p = System.out;  
p.println("hello");
```

### OPTION 2

```
PrintWriter p = new PrintWriter(System.out, true);  
p.println("Hello");  
\public static <T> void p(T s)  
{  
    System.out.println(s);  
}
```

Or this other version:

```
public static void p(Object s)  
{  
    System.out.println(s);  
}
```

As for Eclipse I'm using the suggested shortcut `syso + <Ctrl> + <Space>` :)

```
package some.useful.methods;  
  
public class Print {  
    public static void p(Object s){  
        System.out.println(s);  
    }  
}  
import static some.useful.methods.B.*;  
  
public class A {  
    public static void main(String[] args) {  
        p("Hello!");  
    }  
}
```

## 76 CONCURRENCY (Multi-Threading)

### 76.1 Synchronized vs lock

Lock just locks the object from being used.

Synchronize locks the object AND the entire block must be executed before thread gets interrupted

## 77 COLLECTIONS

include the threadsafe, immutable and collections that include type checking at runtime

## 78 Lambda expressions

# 79 Modules

## 80 JAVADOC

### 80.1.1.1 Writing API Specifications

Ideally, the Java API Specification comprises all assertions required to do a clean-room implementation of the Java Platform for "write once, run anywhere" -- such that any Java applet or application will run the same on any implementation. This may include assertions in the doc comments plus those in any architectural and functional specifications (usually written in FrameMaker) or in any other document. This definition is a lofty goal and there is some practical limitation to how fully we can specify the API. The following are guiding principles we try to follow:

**The Java Platform API Specification is defined by the documentation comments in the source code and any documents marked as specifications reachable from those comments.**

Notice that the specification does not need to be entirely contained in doc comments. In particular, specifications that are lengthy are sometimes best formatted in a separate file and linked to from a doc comment.

**The Java Platform API Specification is a contract between callers and implementations.**

The Specification describes all aspects of the behavior of each method on which a caller can rely. It does not describe implementation details, such as whether the method is native or synchronized. The specification should describe (textually) the thread-safety guarantees provided by a given object. In the absence of explicit indication to the contrary, all objects are assumed to be "thread-safe" (i.e., it is permissible for multiple threads to access them concurrently). It is recognized that current specifications don't always live up to this ideal.

**Unless otherwise noted, the Java API Specification assertions need to be implementation-independent. Exceptions must be set apart and prominently marked as such.**

We have guidelines for [how to prominently document implementation differences](#).

**The Java API Specification should contain assertions sufficient to enable Software Quality Assurance to write complete Java Compatibility Kit (JCK) tests.**

This means that the doc comments must satisfy the needs of the conformance testing by SQA. The comments should not document bugs or how an implementation that is currently out of spec happens to work.

### 80.1.1.2 Writing Programming Guide Documentation

What separates API specifications from a programming guide are examples, definitions of common programming terms, certain conceptual overviews (such as metaphors), and descriptions of implementation bugs and workarounds. There is no dispute that these contribute to a developer's understanding and help a developer write reliable applications more quickly. However, because these do not contain API "assertions", they are not necessary in an API specification. You can include any or all of this information in documentation comments (and can include [custom tags](#), handled by a custom doclet, to facilitate it). At Java Software, we consciously do not include this level of documentation in doc comments, and instead include either links to this information (links to the Java Tutorial and list of changes) or include this information in the same documentation download bundle as the API spec -- the JDK documentation bundle includes the API specs as well as demos, examples, and programming guides.

It's useful to go into further detail about how to document bugs and workarounds. There is sometimes a discrepancy between how code *should* work and how it actually works. This can take two different forms: API spec bugs and code bugs. It's useful to decide up front whether you want to document these in the doc comments. At Java Software we have decided to document both of these outside of doc comments, though we do make exceptions.

**API spec bugs** are bugs that are present in the method declaration or in the doc comment that affects the syntax or semantics. An example of such a spec bug is a method that is specified to throw a `NullPointerException` when `null` is passed in, but `null` is actually a useful parameter that should be accepted (and was even implemented that way). If a decision is made to correct the API specification, it would be useful to state that either in the API specification itself, or in a list of changes to the spec, or both. Documenting an API difference like this in a doc comment, along with its workaround, alerts a developer to the change where they are most likely to see it. Note that an API specification with this correction would still maintain its implementation-independence.

**Code bugs** are bugs in the implementation rather than in the API specification. Code bugs and their workarounds are often likewise distributed separately in a bug report. However, if the Javadoc tool is being used to generate documentation for a particular implementation, it would be quite useful to include this information in the doc comments, suitably separated as a note or by a custom tag (say `@bug`).

### 80.1.1.3 Who Owns and Edits the Doc Comments

The doc comments for the Java platform API specification is owned programmers. However, they are edited by both programmers and writers. It is a basic premise that writers and programmers honor each other's capabilities and both contribute to the best doc comments possible. Often it is a matter of negotiation to determine who writes which parts of the documentation, based on knowledge, time, resources, interest, API complexity, and on the state of the implementation itself. But the final comments must be approved by the responsible engineer.

Ideally, the person designing the API would write the API specification in skeleton source files, with only declarations and doc comments, filling in the implementation only to satisfy the written API contract. The purpose of an API writer is to relieve the



designer from some of this work. In this case, the API designer would write the initial doc comments using sparse language, and then the writer would review the comments, refine the content, and add tags.

If the doc comments are an API specification for re-implementors, and not simply a guide for developers, they should be written either by the programmer who designed and implemented the API, or by a API writer who is or has become a subject matter expert. If the implementation is written to spec but the doc comments are unfinished, a writer can complete the doc comments by inspecting the source code or writing programs that test the API. A writer might inspect or test for exceptions thrown, parameter boundary conditions, and for acceptance of null arguments. However, a much more difficult situation arises if the implementation is *not* written to spec. Then a writer can proceed to write an API specification only if they either know the intent of the designer (either through design meetings or through a separately-written design specification) or have ready access to the designer with their questions. Thus, it may be more difficult for a writer to write the documentation for interfaces and abstract classes that have no implementors.

With that in mind, these guidelines are intended to describe the finished documentation comments. They are intended as *suggestions* rather than requirements to be slavishly followed if they seem overly burdensome, or if creative alternatives can be found. When a complex system such as Java (which contains about 60 packages) is being developed, often a group of engineers contributing to a particular set of packages, such as `javax.swing` may develop guidelines that are different from other groups. This may be due to the differing requirements of those packages, or because of resource constraints.

## Terminology

### 80.1.1.4 API documentation (API docs) or API specifications (API specs)

On-line or hardcopy descriptions of the API, intended primarily for programmers writing in Java. These can be generated using the Javadoc tool or created some other way. An API specification is a particular kind of API document, as described [above](#). An example of an API specification is the on-line [Java Platform, Standard Edition 7 API Specification](#).

### 80.1.1.5 Documentation comments (doc comments)

The special comments in the Java source code that are delimited by the `/** ... */` delimiters. These comments are processed by the Javadoc tool to generate the API docs.

### 80.1.1.6 javadoc

The JDK tool that generates API documentation from documentation comments.

### 80.1.1.7 Source Files

The Javadoc tool can generate output originating from four different types of "source" files:

- Source code files for Java classes (.java) - these contain class, interface, field, constructor and method comments.
- Package comment files - these contain package comments
- Overview comment files - these contain comments about the set of packages
- Miscellaneous unprocessed files - these include images, sample source code, class files, applets, HTML files, and whatever else you might want to reference from the previous files.

For more details, see: [Source Files](#).

### 80.1.1.8 Writing Doc Comments

## Format of a Doc Comment

A doc comment is written in HTML and must precede a class, field, constructor or method declaration. It is made up of two parts -- a description followed by block tags. In this example, the block tags are `@param`, `@return`, and `@see`.

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */
```

```

public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}

```

**Notes:**

- The [resulting HTML](#) from running Javadoc is shown below
- Each line above is indented to align with the code below the comment.
- The first line contains the begin-comment delimiter ( `/**` ).
- Starting with Javadoc 1.4, the [leading asterisks are optional](#).
- Write the first sentence as a short summary of the method, as Javadoc automatically places it in the method summary table (and index).
- Notice the inline tag `{@link URL}`, which converts to an HTML hyperlink pointing to the documentation for the URL class. This inline tag can be used anywhere that a comment can be written, such as in the text following block tags.
- If you have more than one paragraph in the doc comment, separate the paragraphs with a `<p>` paragraph tag, as shown.
- Insert a blank comment line between the description and the list of tags, as shown.
- The first line that begins with an "@" character ends the description. There is only one description block per doc comment; you cannot continue the description following block tags.
- The last line contains the end-comment delimiter ( `*/` ) Note that unlike the begin-comment delimiter, the end-comment contains only a single asterisk.

For more examples, see [Simple Examples](#).

So lines won't wrap, limit any doc-comment lines to 80 characters.

Here is what the previous example would look like after running the Javadoc tool:

**OUTPUT****80.1.1.9 getImage**

```

public Image getImage(URL url,
    String name)

```

Returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute URL. The `name` argument is a specifier that is relative to the `url` argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

**Parameters:**

`url` - an absolute URL giving the base location of the image.  
`name` - the location of the image, relative to the `url` argument.

**Returns:**

the image at the specified URL.

**See Also:**

`Image`

Also see [Troubleshooting Curly Quotes \(Microsoft Word\)](#) at the end of this document.

**Doc Comment Checking Tool**

At Oracle, we have developed a tool for checking doc comments, called the Oracle Doc Check Doclet, or [DocCheck](#). You run it on source code and it generates a report describing what style and tag errors the comments have, and recommends changes. We have tried to make its rules conform to the rules in this document.

DocCheck is a Javadoc doclet, or "plug-in", and so requires that the Javadoc tool be installed (as part of the Java 2 Standard Edition SDK).

**Descriptions****80.1.1.10 First Sentence**

The first sentence of each doc comment should be a summary sentence, containing a concise but complete description of the API item. This means the first sentence of each member, class, interface or package description. The Javadoc tool copies this first sentence to the appropriate member, class/interface or package summary.

This makes it important to write crisp and informative initial sentences that can stand on their own.

This sentence ends at the first period that is followed by a blank, tab, or line terminator, or at the first tag (as defined below). For example, this first sentence ends at "Prof.":

```
/**
 * This is a simulation of Prof. Knuth's MIX computer.
 */
```

However, you can work around this by typing an HTML meta-character such as "&" or "<" immediately after the period, such as:

```
/**
 * This is a simulation of Prof.&nbsp;Knuth's MIX computer.
 */
```

or

```
/**
 * This is a simulation of Prof.<!-- --> Knuth's MIX computer.
 */
```

In particular, write summary sentences that distinguish overloaded methods from each other. For example:

```
/**
 * Class constructor.
 */
foo() {
    ...

    /**
     * Class constructor specifying number of objects to create.
     */
    foo(int n) {
        ...
    }
}
```

#### 80.1.1.11 Implementation-Independence

Write the description to be implementation-independent, but specifying such dependencies where necessary. This helps engineers write code to be "write once, run anywhere."

As much as possible, write doc comments as an implementation-independent API specification.

Define clearly what is required and what is allowed to vary across platforms/implementations.

Ideally, make it complete enough for conforming implementors. Realistically, include enough description so that someone reading the source code can write a substantial suite of conformance tests. Basically, the spec should be complete, including boundary conditions, parameter ranges and corner cases.

Where appropriate, mention what the specification leaves unspecified or allows to vary among implementations.

If you must document implementation-specific behavior, please document it in a separate paragraph with a lead-in phrase that makes it clear it is implementation-specific. If the implementation varies according to platform, then specify "On <platform>" at the start of the paragraph. In other cases that might vary with implementations on a platform you might use the lead-in phrase "Implementation-Specific:". Here is an example of an implementation-dependent part of the specification for `java.lang.Runtime`:

On Windows systems, the path search behavior of the `loadLibrary` method is identical to that of the Windows API's `LoadLibrary` procedure.

The use of "On Windows" at the beginning of the sentence makes it clear up front that this is an implementation note.

#### 80.1.1.12 Automatic re-use of method comments

You can avoid re-typing doc comments by being aware of how the Javadoc tool duplicates (inherits) comments for methods that override or implement other methods. This occurs in three cases:

- When a method in a class overrides a method in a superclass
- When a method in an interface overrides a method in a superinterface
- When a method in a class implements a method in an interface

In the first two cases, if a method `m()` overrides another method, The Javadoc tool will generate a subheading "Overrides" in the documentation for `m()`, with a link to the method it is overriding.

In the third case, if a method `m()` in a given class implements a method in an interface, the Javadoc tool will generate a subheading "Specified by" in the documentation for `m()`, with a link to the method it is implementing.

In all three of these cases, if the method `m()` contains no doc comments or tags, the Javadoc tool will also copy the text of the method it is overriding or implementing to the generated documentation for `m()`. So if the documentation of the overridden or implemented method is sufficient, you do not need to add documentation for `m()`. If you add any documentation comment or tag to `m()`, the "Overrides" or "Specified by" subheading and link will still appear, but no text will be copied.

## A Style Guide

The following are useful tips and conventions for writing descriptions in doc comments.

### Use `<code>` style for keywords and names.

Keywords and names are offset by `<code> . . . </code>` when mentioned in a description. This includes:

- Java keywords
- package names
- class names
- method names
- interface names
- field names
- argument names
- code examples

#### 80.1.1.13 Use in-line links economically

You are encouraged to add links for API names (listed immediately above) using the `{@link}` tag. It is not necessary to add links for *all* API names in a doc comment. Because links call attention to themselves (by their color and underline in HTML, and by their length in source code doc comments), it can make the comments more difficult to read if used profusely. We therefore recommend adding a link to an API name if:

The user might actually want to click on it for more information (in your judgment), and

Only for the first occurrence of each API name in the doc comment (don't bother repeating a link)

Our audience is advanced (not novice) programmers, so it is generally not necessary to link to API in the `java.lang` package (such as `String`), or other API you feel would be well-known.

#### 80.1.1.14 Omit parentheses for the general form of methods and constructors

When referring to a method or constructor that has multiple forms, and you mean to refer to a specific form, use parentheses and argument types. For example, `ArrayList` has two `add` methods: `add(Object)` and `add(int, Object)`:

The `add(int, Object)` method adds an item at a specified position in this arraylist.

However, if referring to both forms of the method, omit the parentheses altogether. It is misleading to include empty parentheses, because that would imply a particular form of the method. The intent here is to distinguish the general method from any of its particular forms. Include the word "method" to distinguish it as a method and not a field.

The `add` method enables you to insert items. (preferred)

The `add()` method enables you to insert items. (avoid when you mean "all forms" of the `add` method)

#### 80.1.1.15 OK to use phrases instead of complete sentences, in the interests of brevity.

This holds especially in the initial summary and in @param tag descriptions.

#### 80.1.1.16 Use 3rd person (descriptive) not 2nd person (prescriptive).

The description is in 3rd person declarative rather than 2nd person imperative.

Gets the label. (preferred)

Get the label. (avoid)

#### 80.1.1.17 Method descriptions begin with a verb phrase.

A method implements an operation, so it usually starts with a verb phrase:

Gets the label of this button. (preferred)

This method gets the label of this button.

#### 80.1.1.18 Class/interface/field descriptions can omit the subject and simply state the object.

These API often describe things rather than actions or behaviors:

A button label. (preferred)

This field is a button label. (avoid)

#### 80.1.1.19 Use "this" instead of "the" when referring to an object created from the current class.

For example, the description of the `getToolkit` method should read as follows:

Gets the toolkit for this component. (preferred)

Gets the toolkit for the component. (avoid)

#### 80.1.1.20 Add description beyond the API name.

The best API names are "self-documenting", meaning they tell you basically what the API does. If the doc comment merely repeats the API name in sentence form, it is not providing more information. For example, if method description uses only the words that appear in the method name, then it is adding nothing at all to what you could infer. The ideal comment goes beyond those words and should always reward you with some bit of information that was not immediately obvious from the API name.

**Avoid** - The description below says nothing beyond what you know from reading the method name. The words "set", "tool", "tip", and "text" are simply repeated in a sentence.

```
/**
 * Sets the tool tip text.
 *
 * @param text the text of the tool tip
 */
public void setToolTipText(String text) {
```

**Preferred** - This description more completely defines what a tool tip is, in the larger context of registering and being displayed in response to the cursor.

```
/**
 * Registers the text to display in a tool tip. The text
 * displays when the cursor lingers over the component.
 *
 * @param text the string to display. If the text is null,
 * the tool tip is turned off for this component.
 */
public void setToolTipText(String text) {
```

#### Be clear when using the term "field".

Be aware that the word "field" has two meanings:

- static field, which is another term for "class variable"
- text field, as in the `TextField` class. Note that this kind of field might be restricted to holding dates, numbers or any text. Alternate names might be "date field" or "number field", as appropriate.

**Avoid Latin abbreviations**

use "also known as" instead of "aka", use "that is" or "to be specific" instead of "i.e.", use "for example" instead of "e.g.", and use "in other words" or "namely" instead of "viz."

**Tag Conventions**

This section covers:

- [Order of Tags](#)
- [Ordering Multiple Tags](#)
- [Required Tags](#)
- [Tag](#)

Comments: [@author](#) [@version](#) [@param](#) [@return](#) [@deprecated](#) [@since](#) [@throws](#) [@exception](#) [@see](#) [@serial](#) [@serialField](#) [@serialData](#) [{@link}](#)

- [Custom tags and Annotations](#)

Most of the following tags are specified in the [Java Language Specification, First Edition](#). Also see the [Javadoc reference page](#).

**80.1.1.21 Order of Tags**

Include tags in the following order:

- [@author](#) (classes and interfaces only, required)
- [@version](#) (classes and interfaces only, required. See [footnote 1](#))
- [@param](#) (methods and constructors only)
- [@return](#) (methods only)
- [@exception](#) ([@throws](#) is a synonym added in Javadoc 1.2)
- [@see](#)
- [@since](#)
- [@serial](#) (or [@serialField](#) or [@serialData](#))
- [@deprecated](#) (see [How and When To Deprecate APIs](#))

**80.1.1.22 Ordering Multiple Tags**

We employ the following conventions when a tag appears more than once in a documentation comment. If desired, groups of tags, such as multiple [@see](#) tags, can be separated from the other tags by a blank line with a single asterisk.

Multiple [@author](#) tags should be listed in chronological order, with the creator of the class listed at the top.

Multiple [@param](#) tags should be listed in argument-declaration order. This makes it easier to visually match the list to the declaration.

Multiple [@throws](#) tags (also known as [@exception](#)) should be listed alphabetically by the exception names.

Multiple [@see](#) tags should be ordered as follows, which is roughly the same order as their arguments are [searched for by javadoc](#), basically from nearest to farthest access, from least-qualified to fully-qualified. The following list shows this progression. Notice the methods and constructors are in "telescoping" order, which means the "no arg" form first, then the "1 arg" form, then the "2 arg" form, and so forth. Where a second sorting key is needed, they could be listed either alphabetically or grouped logically.

```
@see #field
@see #Constructor(Type, Type...)
@see #Constructor(Type id, Type id...)
@see #method(Type, Type,...)
@see #method(Type id, Type, id...)
@see Class
@see Class#field
@see Class#Constructor(Type, Type...)
@see Class#Constructor(Type id, Type id)
@see Class#method(Type, Type,...)
@see Class#method(Type id, Type id,...)
@see package.Class
@see package.Class#field
@see package.Class#Constructor(Type, Type...)
@see package.Class#Constructor(Type id, Type id)
@see package.Class#method(Type, Type,...)
@see package.Class#method(Type id, Type, id)
@see package
```

**80.1.1.23 Required Tags**

An `@param` tag is "required" (by convention) for every parameter, even when the description is obvious. The `@return` tag is required for every method that returns something other than `void`, even if it is redundant with the method description. (Whenever possible, find something non-redundant (ideally, more specific) to use for the tag comment.)

These principles expedite automated searches and automated processing. Frequently, too, the effort to avoid redundancy pays off in extra clarity.

#### 80.1.1.24 Tag Comments

As a reminder, the fundamental use of these tags is described on the [Javadoc Reference page](#). Java Software generally uses the following additional guidelines to create comments for each tag:

##### `@author` ([reference page](#))

You can provide one `@author` tag, multiple `@author` tags, or no `@author` tags. In these days of the community process when development of new APIs is an open, joint effort, the JSR can be consider the author for new packages at the package level. For example, the new package `java.nio` has "`@author JSR-51 Expert Group`" at the package level. Then individual programmers can be assigned to `@author` at the class level. As this tag can only be applied at the overview, package and class level, the tag applies only to those who make significant contributions to the design or implementation, and so would not ordinarily include technical writers.

The `@author` tag is not critical, because it is not included when generating the API specification, and so it is seen only by those viewing the source code. (Version history can also be used for determining contributors for internal purposes.)

If someone felt strongly they need to add `@author` at the member level, they could do so by running javadoc using the new 1.4 – tag option:

```
-tag author:a:"Author:"
```

If the author is unknown, use "unascrbed" as the argument to `@author`. Also see [order of multiple @author tags](#).

##### `@version` ([reference page](#))

The Java Software convention for the argument to the `@version` tag is the SCCS string "`%I%, %G%`", which converts to something like "`1.39, 02/28/97`" (mm/dd/yy) when the file is checked out of SCCS.

##### `@param` ([reference page](#))

The `@param` tag is followed by the name (not data type) of the parameter, followed by a description of the parameter. By convention, the first noun in the description is the data type of the parameter. (Articles like "a", "an", and "the" can precede the noun.) An exception is made for the primitive `int`, where the data type is usually omitted. Additional spaces can be inserted between the name and description so that the descriptions line up in a block. Dashes or other punctuation should not be inserted before the description, as the Javadoc tool inserts one dash.

Parameter names are lowercase by convention. The data type starts with a lowercase letter to indicate an object rather than a class. The description begins with a lowercase letter if it is a phrase (contains no verb), or an uppercase letter if it is a sentence. End the phrase with a period only if another phrase or sentence follows it.

Example:

```
* @param ch          the character to be tested
* @param observer    the image observer to be notified
```

Do not bracket the name of the parameter after the `@param` tag with `<code>...</code>` since Javadoc 1.2 and later automatically do this. (Beginning with 1.4, the name cannot contain any HTML, as Javadoc compares the `@param` name to the name that appears in the signature and emits a warning if there is any difference.)

When writing the comments themselves, in general, start with a phrase and follow it with sentences if they are needed.

When writing a phrase, do not capitalize and do not end with a period:

```
@param x    the x-coordinate, measured in pixels
```

When writing a phrase followed by a sentence, do not capitalize the phrase, but end it with a period to distinguish it from the start of the next sentence:

```
@param x    the x-coordinate. Measured in pixels.
```

If you prefer starting with a sentence, capitalize it and end it with a period:

```
@param x    Specifies the x-coordinate, measured in pixels.
```

When writing multiple sentences, follow normal sentence rules:

```
@param x    Specifies the x-coordinate. Measured in pixels.
```

Also see [order of multiple @param tags](#).

##### `@return` ([reference page](#))

Omit `@return` for methods that return `void` and for constructors; include it for all other methods, even if its content is entirely redundant with the method description. Having an explicit `@return` tag makes it easier for someone to find the return value



quickly. Whenever possible, supply return values for special cases (such as specifying the value returned when an out-of-bounds argument is supplied).

Use the same capitalization and punctuation as you used in @param.

### @deprecated (reference page)

The @deprecated description in the first sentence should at least tell the user when the API was deprecated and what to use as a replacement. Only the first sentence will appear in the summary section and index. Subsequent sentences can also explain why it has been deprecated. When generating the description for a deprecated API, the Javadoc tool moves the @deprecated text ahead of the description, placing it in italics and preceding it with a bold warning: "Deprecated". An @see tag (for Javadoc 1.1) or {@link} tag (for Javadoc 1.2 or later) should be included that points to the replacement method:

For Javadoc 1.2 and later, the standard format is to use @deprecated tag and the in-line {@link} tag. This creates the link in-line, where you want it. For example:

```
/**
 * @deprecated As of JDK 1.1, replaced by
 *             {@link #setBounds(int,int,int,int)}
 */
```

For Javadoc 1.1, the standard format is to create a pair of @deprecated and @see tags. For example:

```
/**
 * @deprecated As of JDK 1.1, replaced by
 *
 *             setBounds
 * @see #setBounds(int,int,int,int)
 */
```

If the member has no replacement, the argument to @deprecated should be "No replacement".

Do not add @deprecated tags without first checking with the appropriate engineer. Substantive modifications should likewise be checked first.

### @since (reference page)

Specify the product version when the Java name was added to the API specification (if different from the implementation). For example, if a package, class, interface or member was added to the Java 2 Platform, Standard Edition, API Specification at version 1.2, use:

```
/**
 * @since 1.2
 */
```

The Javadoc standard doclet displays a "Since" subheading with the string argument as its text. This subheading appears in the generated text only in the place corresponding to where the @since tag appears in the source doc comments (The Javadoc tool does not proliferate it down the hierarchy).

(The convention once was " @since JDK1.2" but because this is a specification of the Java Platform, not particular to the Oracle JDK or SDK, we have dropped "JDK".)

When a package is introduced, specify an @since tag in its package description and each of its classes. (Adding @since tags to each class is technically not needed, but is our convention, as enables greater visibility in the source code.) In the absence of overriding tags, the value of the @since tag applies to each of the package's classes and members.

When a class (or interface) is introduced, specify one @since tag in its class description and no @since tags in the members. Add an @since tag only to members added in a later version than the class. This minimizes the number of @since tags.

If a member changes from protected to public in a later release, the @since tag would not change, even though it is now usable by any caller, not just subclasses.

### @throws (@exception was the original tag) (reference page)

A @throws tag should be included for any checked exceptions (declared in the throws clause), as illustrated below, and also for any unchecked exceptions that the caller might reasonably want to catch, with the exception of `NullPointerException`. Errors should not be documented as they are unpredictable. For more details, please see [Documenting Exceptions with the @throws Tag](#).

```
/**
 * @throws IOException If an input or output
 *                   exception occurred
 */
public void f() throws IOException {
```



```
// body
}
```

See the [Exceptions chapter](#) of the *Java Language Specification, Second Edition* for more on exceptions. Also see [order of multiple @throws tags](#).

[@see \(reference page\)](#)

Also see [order of multiple @see tags](#).

[@serial](#)

[@serialField](#)

[@serialData](#)

(All added in Javadoc 1.2) ([reference page](#))

For information about how to use these tags, along with an example, see "Documenting Serializable Fields and Data for a Class," [Section 1.6 of the Java Object Serialization Specification](#). Also see Oracle's [criteria](#) for including classes in the serialized form specification.

[{@link} \(Added in Javadoc 1.2\) \(reference page\)](#)

For conventions, see [Use In-Line Links Economically](#).

### Custom Tags and Annotations

If annotations are new to you, when you need to markup your source code, it might not be immediately clear whether to use an annotation or a Javadoc custom tag. Here is a quick comparison of the two.

In general, if the markup is intended to affect or produce documentation, it should probably be a javadoc tag; otherwise, it should be an annotation.

- Tag - Intended as a way of adding structure and content to the documentation. Allows multi-line text to be provided. (Use the `-tag` or `-taglet` Javadoc option to create custom tags.) Tags should never affect program semantics.
- Annotation - Does not directly affect program semantics, but does affect the way programs are treated by tools and libraries, which can in turn affect the semantics of the running program. Annotations can be read from source files, class files, or reflectively at run time. Allows a single line of text to be provided.

If you need to affect both program semantics and documentation, you probably need both an annotation and a tag. For example, our guidelines now recommend using the `@Deprecated` annotation for alerting the compiler warning and the `@deprecated` tag for the comment text.

## Documenting Default Constructors

[Section 8.8.7 of the Java Language Specification, Second Edition](#) describes a default constructor: If a class contains no constructor declarations, then a default constructor that takes no parameters is automatically provided. It invokes the superclass constructor with no arguments. The constructor has the same access as its class.

The Javadoc tool generates documentation for default constructors. When it documents such a constructor, Javadoc leaves its description blank, because a default constructor can have no doc comment. The question then arises: How do you add a doc comment for a default constructor? The simple answer is that it is not possible -- and, conveniently, our programming convention is to avoid default constructors. (We considered but rejected the idea that the Javadoc tool should generate a default comment for default constructors.)

Good programming practice dictates that code should never make use of default constructors in public APIs: **All constructors should be explicit**. That is, all default constructors in public and protected classes should be turned into explicit constructor declarations with the appropriate access modifier. This explicit declaration also gives you a place to write documentation comments.

The reason this is good programming practice is that an explicit declaration helps prevents a class from inadvertently being made instantiable, as the design engineer has to actually make a decision about the constructor's access. We have had several cases where we did not want a public class to be instantiable, but the programmer overlooked the fact that its default constructor was public. If a class is inadvertently allowed to be instantiable in a released version of a product, upward compatibility dictates that the unintentional constructor be retained in future versions. Under these unfortunate circumstances, the constructor should be made explicit and deprecated (using `@deprecated`).

Note that when creating an explicit constructor, it must match *precisely* the declaration of the automatically generated constructor; even if the constructor should logically be protected, it must be made public to match the declaration of the automatically generated constructor, for compatibility. An appropriate doc comment should then be provided. Often, the comment should be something as simple as:

```
/**
 * Sole constructor. (For invocation by subclass
 * constructors, typically implicit.)
 */
```

```
protected AbstractMap() { }
```

## Documenting Exceptions with @throws Tag

NOTE - The tags `@throws` and `@exception` are synonyms.

### 80.1.1.25 Documenting Exceptions in API Specs

The API specification for methods is a contract between a caller and an implementor. Javadoc-generated API documentation contains two ways of specifying this contract for exceptions -- the "throws" clause in the declaration, and the `@throws` Javadoc tag. These guidelines describe how to document exceptions with the `@throws` tag.

### 80.1.1.26 Throws Tag

The purpose of the `@throws` tag is to indicate which exceptions the programmer must catch (for checked exceptions) or might want to catch (for unchecked exceptions).

### 80.1.1.27 Guidelines - Which Exceptions to Document

Document the following exceptions with the `@throws` tag:

- **All checked exceptions.**  
(These must be declared in the throws clause.)
- **Those unchecked exceptions that the caller might reasonably want to catch.**  
(It is considered poor programming practice to include unchecked exceptions in the throws clause.)  
Documenting these in the `@throws` tag is up to the judgment of the API designer, as described below.

### 80.1.1.28 Documenting Unchecked Exceptions

It is generally desirable to document the unchecked exceptions that a method can throw: this allows (but does not require) the caller to handle these exceptions. For example, it allows the caller to "translate" an implementation-dependent unchecked exception to some other exception that is more appropriate to the caller's exported abstraction.

Since there is no way to guarantee that a call has documented all of the unchecked exceptions that it may throw, the programmer must not depend on the presumption that a method cannot throw any unchecked exceptions other than those that it is documented to throw. In other words, you should always assume that a method can throw unchecked exceptions that are undocumented.

Note that it is always inappropriate to document that a method throws an unchecked exception that is tied to the current implementation of that method. In other words, document exceptions that are independent of the underlying implementation. For example, a method that takes an index and uses an array internally should *not* be documented to throw an `ArrayIndexOutOfBoundsException`, as another implementation could use a data structure other than an array internally. It is, however, generally appropriate to document that such a method throws an `IndexOutOfBoundsException`.

Keep in mind that if you do not document an unchecked exception, other implementations are free to not throw that exception. Documenting exceptions properly is an important part of write-once, run-anywhere.

### 80.1.1.29 Background on Checked and Unchecked Exceptions

The idea behind *checking* an exception is that the compiler checks at compile-time that the exception is properly being caught in a try-catch block.

You can identify checked and unchecked exceptions as follows.

- Unchecked exceptions are the classes `RuntimeException`, `Error` and their subclasses.
- All other exception subclasses are checked exceptions.

Note that whether an exception is checked or unchecked is not defined by whether it is included in a throws clause.

### 80.1.1.30 Background on the Throws Clause

Checked exceptions must be included in a throws clause of the method. This is necessary for the compiler to know which exceptions to check. For example (in `java.lang.Class`):

```
public static Class forName(String className)
    throws ClassNotFoundException
```

By convention, unchecked exceptions should not be included in a throws clause. (Including them is considered to be poor programming practice. The compiler treats them as comments, and does no checking on them.) The following is poor code -- since the exception is a RuntimeException, it should be documented in the @throws tag instead.

java.lang.Byte source code:

```
public static Byte valueOf(String s, int radix)
    throws NumberFormatException
```

Note that the *Java Language Specification* also shows unchecked exceptions in throws clauses (as follows). Using the throws clause for unchecked exceptions in the spec is merely a device meant to indicate this exception is part of the contract between the caller and implementor. The following is an example of this (where "final" and "synchronization" are removed to make the comparison simpler).

java.util.Vector source code:

```
public Object elementAt(int index)
    java.util.Vector spec in the Java Language Specification, 1st Ed. (p. 656):
    public Object elementAt(int index)
        throws IndexOutOfBoundsException
```

## Package-Level Comments

With Javadoc 1.2, package-level doc comments are available. Each package can have its own package-level doc comment source file that The Javadoc tool will merge into the documentation that it produces. This file is named `package.html` (and is same name for all packages). This file is kept in the source directory along with all the `*.java` files. (Do not put the `packages.html` file in the new doc-files source directory, because those files are only copied to the destination and are not processed.)

The file `package.html` is an example of a package-level source file for `java.textcode>` and `package-summary.html` is the file that the Javadoc tool generates.

The Javadoc tool processes `package.html` by doing three things:

Copies its contents (everything between `<body>` and `</body>`) below the summary tables in the destination file `package-summary.html`.

Processes any @see, @since or {@link} Javadoc tags that are present.

Copies the first sentence to the right-hand column of the [Overview Summary](#).

### 80.1.1.31 Template for package.html source file

At Oracle, we use the following template, [Empty Template for Package-Level Doc Comment File](#), when creating a new package doc comment file. This contains a copyright statement. Obviously, if you are from a different company, you would supply your own copyright statement. An engineer would copy this whole file, rename it to `package.html`, and delete the lines set off with hash marks:#####. One such file should go into each package directory of the source tree.

### 80.1.1.32 Contents of package.html source file

The package doc comment should provide (directly or via links) everything necessary to allow programmers to use the package. It is a very important piece of documentation: for many facilities (those that reside in a single package but not in a single class) it is the first place where programmers will go for documentation. It should contain a short, readable description of the facilities provided by the package (in the introduction, below) followed by pointers to detailed documentation, or the detailed documentation itself, whichever is appropriate. Which is appropriate will depend on the package: a pointer is appropriate if it's part of a larger system (such as, one of the 37 packages in Corba), or if a Framemaker document already exists for the package; the detailed documentation should be contained in the package doc comment file itself if the package is self-contained and doesn't require extensive documentation (such as `java.math`).

To sum up, the primary purpose of the package doc comment is to describe the purpose of the package, the conceptual framework necessary to understand and to use it, and the relationships among the classes that comprise it. For large, complex packages (and those that are part of large, complex APIs) a pointer to an external architecture document is warranted.

The following are the sections and headings you should use when writing a package-level comment file. There should be no heading before the first sentence, because the Javadoc tool picks up the first text as the summary statement.

Make the first sentence a summary of the package. For example: "Provides classes and interfaces for handling text, dates, numbers and messages in a manner independent of natural languages."

Describe what the package contains and state its purpose.

### 80.1.1.33 Package Specification

**Include a description of or links to any package-wide specifications for this package that are not included in the rest of the javadoc-generated documentation.** For example, the java.awt package might describe how the general behavior in that package is allowed to vary from one operating system to another (Windows, Solaris, Mac).

**Include links to any specifications written outside of doc comments (such as in FrameMaker or whatever) if they contain *assertions* not present in the javadoc-generated files.**

An *assertion* is a statement a conforming implementor would have to know in order to implement the Java platform.

On that basis, at Oracle, references in this section are critical to the Java Compatibility Kit (JCK). The Java Compatibility Kit includes a test to verify each assertion, to determine what passes as Java Compatible. The statement "Returns an int" is an assertion. An example is not an assertion.

Some "specifications" that engineers have written contain no assertions not already stated in the API specs (javadoc) -- they just elaborate on the API specs. In this respect, such a document should not be referred to in this section, but rather should be referred to in the next section.

#### 80.1.1.34 Include specific references.

If only a section of a referenced document should be considered part of the API spec, then you should link or refer to only that section and refer to the rest of the document in the next section. The idea is to clearly delineate what is part of the API spec and what is not, so the JCK team can write tests with the proper breadth. This might even encourage some writers to break documents apart so specs are separate.

#### 80.1.1.35 Related Documentation

Include references to any documents that do *not* contain specification assertions, such as overviews, tutorials, examples, demos, and guides.

#### 80.1.1.36 Class and Interface Summary

[Omit this section until we implement @category tag]

Describe logical groupings of classes and interfaces

@see other packages, classes and interfaces

## Documenting Anonymous Inner Classes

Anonymous inner classes are defined in Java Language Specification, Second Edition, at [Anonymous Class Declaration](#). The Javadoc tool does not directly document anonymous classes -- that is, their declarations and doc comments are ignored. If you want to document an anonymous class, the proper way to do so is in a doc comment of its outer class, or another closely associated class.

For example, if you had an anonymous `TreeSelectionListener` inner class in a method `makeTree` that returns a `JTree` object that users of this class might want to override, you could document in the method comment that the returned `JTree` has a `TreeSelectionListener` attached to it:

```
/**
 * The method used for creating the tree. Any structural
 * modifications to the display of the Jtree should be done
 * by overriding this method.
 * <p>
 * This method adds an anonymous TreeSelectionListener to
 * the returned JTree. Upon receiving TreeSelectionEvents,
 * this listener calls refresh with the selected node as a
 * parameter.
 */
public JTree makeTree(AreaInfo ai){
}
```

## Including Images

This section covers images used in the doc comments, not images directly used by the source code.

NOTE: The bullet and heading images required with Javadoc version 1.0 and 1.1 are located in the images directory of the JDK download bundle: `jdk1.1/docs/api/images/`. Those images are no longer needed starting with 1.2.

Prior to Javadoc 1.2, the Javadoc tool would not copy images to the destination directory -- you had to do it in a separate operation, either manually or with a script. Javadoc 1.2 looks for and copies to the destination directory a directory named "doc-files" in the source tree (one for each package) and its contents. (It does a shallow copy for 1.2 and 1.3, and a deep copy for 1.4

and later.) Thus, you can put into this directory any images (GIF, JPEG, etc) or other files not otherwise processed by the Javadoc tool.

The following are the Java Software proposals for conventions for including images in doc comments. The master images would be located in the source tree; when the Javadoc tool is run with the standard doclet, it would copy those files to the destination HTML directory.

#### 80.1.1.37 Images in Source Tree

**Naming of doc images in source tree** - Name GIF images `<class>-1.gif`, incrementing the integer for subsequent images in the same class. Example:

`Button-1.gif`

**Location of doc images in source tree** - Put doc images in a directory called "doc-files". This directory should reside in the same package directory where the source files reside. (The name "doc-files" distinguishes it as documentation separate from images used by the source code itself, such as bitmaps displayed in the GUI.) Example: A screen shot of a button, `Button-1.gif`, might be included in the class comment for the Button class. The Button source file and the image would be located at:

`java/awt/Button.java` (source file)

`java/awt/doc-files/Button-1.gif` (image file)

#### 80.1.1.38 Images in HTML Destination

##### 80.1.1.39 Naming of doc images in HTML destination -

Images would have the same name as they have in the source tree. Example:

`Button-1.gif`

##### 80.1.1.40 Location of doc images in HTML destination

With hierarchical file output, such as Javadoc 1.2, directories would be located in the package directory named "doc-files". For example:

`api/java/awt/doc-files/Button-1.gif`

With flat file output, such as Javadoc 1.1, directories would be located in the package directory and named "images-<package>". For example:

- o `api/images-java.awt/`
- `api/images-java.awt.swing/`

## Examples of Doc Comments

```
/**
 * Graphics is the abstract base class for all graphics contexts
 * which allow an application to draw onto components realized on
 * various devices or onto off-screen images.
 * A Graphics object encapsulates the state information needed
 * for the various rendering operations that Java supports. This
 * state information includes:
 * <ul>
 * <li>The Component to draw on
 * <li>A translation origin for rendering and clipping coordinates
 * <li>The current clip
 * <li>The current color
 * <li>The current font
 * <li>The current logical pixel operation function (XOR or Paint)
 * <li>The current XOR alternation color
 * (see <a href="#setXORMode">setXORMode</a>)
 * </ul>
 * <p>
 * Coordinates are infinitely thin and lie between the pixels of the
 * output device.
 * Operations which draw the outline of a figure operate by traversing
 * along the infinitely thin path with a pixel-sized pen that hangs
 * down and to the right of the anchor point on the path.
 * Operations which fill a figure operate by filling the interior
 * of the infinitely thin path.
 * Operations which render horizontal text render the ascending
 * portion of the characters entirely above the baseline coordinate.
```

```

* <p>
* Some important points to consider are that drawing a figure that
* covers a given rectangle will occupy one extra row of pixels on
* the right and bottom edges compared to filling a figure that is
* bounded by that same rectangle.
* Also, drawing a horizontal line along the same y coordinate as
* the baseline of a line of text will draw the line entirely below
* the text except for any descenders.
* Both of these properties are due to the pen hanging down and to
* the right from the path that it traverses.
* <p>
* All coordinates which appear as arguments to the methods of this
* Graphics object are considered relative to the translation origin
* of this Graphics object prior to the invocation of the method.
* All rendering operations modify only pixels which lie within the
* area bounded by both the current clip of the graphics context
* and the extents of the Component used to create the Graphics object.
*
* @author      Sami Shaio
* @author      Arthur van Hoff
* @version     %I%, %G%
* @since      1.0
*/
public abstract class Graphics {

    /**
     * Draws as much of the specified image as is currently available
     * with its northwest corner at the specified coordinate (x, y).
     * This method will return immediately in all cases, even if the
     * entire image has not yet been scaled, dithered and converted
     * for the current output device.
     * <p>
     * If the current output representation is not yet complete then
     * the method will return false and the indicated
     * {@link ImageObserver} object will be notified as the
     * conversion process progresses.
     *
     * @param img      the image to be drawn
     * @param x        the x-coordinate of the northwest corner
     *                 of the destination rectangle in pixels
     * @param y        the y-coordinate of the northwest corner
     *                 of the destination rectangle in pixels
     * @param observer the image observer to be notified as more
     *                 of the image is converted. May be
     *                 <code>null</code>
     * @return         <code>true</code> if the image is completely
     *                 loaded and was painted successfully;
     *                 <code>false</code> otherwise.
     * @see            Image
     * @see            ImageObserver
     * @since          1.0
     */
    public abstract boolean drawImage(Image img, int x, int y,
                                     ImageObserver observer);

    /**
     * Dispose of the system resources used by this graphics context.
     * The Graphics context cannot be used after being disposed of.
     * While the finalization process of the garbage collector will
     * also dispose of the same system resources, due to the number
     * of Graphics objects that can be created in short time frames

```



```

    * it is preferable to manually free the associated resources
    * using this method rather than to rely on a finalization
    * process which may not happen for a long period of time.
    * <p>
    * Graphics objects which are provided as arguments to the paint
    * and update methods of Components are automatically disposed
    * by the system when those methods return. Programmers should,
    * for efficiency, call the dispose method when finished using
    * a Graphics object only if it was created directly from a
    * Component or another Graphics object.
    *
    * @see      #create(int, int, int, int)
    * @see      #finalize()
    * @see      Component#getGraphics()
    * @see      Component#paint(Graphics)
    * @see      Component#update(Graphics)
    * @since    1.0
    */
    public abstract void dispose();

    /**
     * Disposes of this graphics context once it is no longer
     * referenced.
     *
     * @see      #dispose()
     * @since    1.0
     */
    public void finalize() {
        dispose();
    }
}

```

### Troubleshooting Curly Quotes (Microsoft Word)

**Problem** - A problem occurs if you are working in an editor that defaults to curly (rather than straight) single and double quotes, such as Microsoft Word on a PC -- the quotes disappear when displayed in some browsers (such as Unix Netscape). So a phrase like "the display's characteristics" becomes "the displays characteristics."

The illegal characters are the following:

- 146 - right single quote
- 147 - left double quote
- 148 - right double quote

What should be used instead is:

- 39 - straight single quote
- 34 - straight quote

**Preventive Solution** - The reason the "illegal" quotes occurred was that a default Word option is "Change 'Straight Quotes' to 'Smart Quotes'". If you turn this off, you get the appropriate straight quotes when you type.

**Fixing the Curly Quotes** - Microsoft Word has several save options -- use "Save As Text Only" to change the quotes back to straight quotes. Be sure to use the correct option:

- "Save As Text Only With Line Breaks" - inserts a space at the end of each line, and keeps curly quotes.
- "Save As Text Only" - does not insert a space at the end of each lines, and changes curly quotes to straight quotes.