Java 5 & 6 Reference Card

Java is a trademark of Sun Microsystems

Comments

//	single-line comment extends to end-of-line	
/*	multi-line comment extends until terminated by:	*/
/**	javadoc multi-line comment extends until terminated by:	*/
	Usually start intermediate lines in multi-line comments with:	*

javadoc: @param @return @throws/@exception @see @serialField
@author @version @since @deprecated @serial @serialData

Built-In Types & Wrappers and Indirect References

	bits	*in java.lang*	*in java.util.concurrent.atomic*	
char	16	Character		
boolean	1	Boolean	AtomicBoolean	
long	64	Long	AtomicLong	*numeric wrapper classes all extend java.lang.*Number
int	32	Integer	AtomicInteger	
short	16	Short		
byte	8	Byte		
	∞	*java.math.*BigInteger		
float	32	Float *IEEE 754-1985*		
double	64	Double *IEEE 754-1985*		
	∞	*java.math.*BigDecimal		

java.lang.ref.(Phantom|Soft|Weak)Reference extend **Reference** for coordination with GC: can **get**, **clear** value, **enqueue** in a **ReferenceQueue**, check if **isEnqueued**

Literal Values of Standard Types

boolean:	**true**, **false**	ref types:	**null**, **this**, **super**
Boolean:	TRUE, FALSE		
int:	255, 0xff, 0377	long:	365l, 0x2feL
double:	1.2, 6.54e21, 3.7E-4	float:	1.2f, 1e-9F
Double, Float:	NaN, POSITIVE_INFINITY, NEGATIVE_INFINITY, MAX_VALUE, MIN_VALUE		
char:	'A', '\b', '\t', '\n', '\f', '\r', '\"', '\'', '\\', '\xxx', '\uxxxx'		

Declarations (except Arrays, Enums, Generics)

Package membership (must appear at top of file): **package** package-name**;**

Accessing other packages: **import** [**static**] package-name**.∗;**
import [**static**] package-name**.**class-name**;**

Interface: **interface** identifier [**extends** interface-id [**,** interface-id]*]
{ [constants]* [method-signature**;**]* }

Class: **class** identifier [**extends** class-id]
[**implements** interface-id [**,** interface-id]*]
{ [field]* [instance-initializer]* [method]* [class-initializer]* }

NB: multiple inheritance of interfaces, single inheritance of classes;
fields & methods are members of instances or the class; methods may be **void**.
new class-id **(** [args] **)**, **this (** [args] **)** & **super (** [args] **)** invoke constructors on classes; interface, abstract class and class names can all be used as types

Constructor Method Signature: class-id **(** [parameter [**,** parameter]*])
[**throws** throwable-id [**,** throwable-id]*]

Instance Method Signature: type identifier **(** [parameter [**,** parameter]*])
[**throws** throwable-id [**,** throwable-id]*]

Class Method Signature: **static** type identifier **(** [parameter [**,** parameter]*])
[**throws** throwable-id [**,** throwable-id]*]

Method Parameter: type identifier

NB: Final *vararg* parameter: type**...** identifier ≈ type **[ ]** identifier

Method: method-signature { [statement]* }

Instance Field(s): type identifier [= expr] [**,** identifier [= expr]]* **;**

Class Field(s): **static** type identifier [= expr] [**,** identifier [= expr]]* **;**

Local Variable(s): type identifier [= expr] [**,** identifier [= expr]]* **;**

Constant(s): **static final** type identifier = expr [**,** identifier = expr]* **;**

Arrays

Declaration: basetype [**[ ]**]+ id [= array-expr] [**,** id [= array-expr]]*

Construction: **new** basetype [**[***size***]**]+ *Literal:* { expr [**,** expr]* }

Initializer: **new** basetype [**[ ]**]+ { expr [**,** expr]* }

Slot access: array-variable [**[***index***]**]+ 0 ≤ *index* < array**.length**

E.g. int **[ ] [ ]** a = { { 0 }, { 0,1 } }**;** b = new int **[5] [7];** b **[2] [3]** = 8+a **[1] [1];**

java.util.Arrays: sort binarySearch fill [deep](equals|hashCode|toString)

Declaration Modifiers

Implementation incomplete, cannot instantiate (class, method):	**abstract**
Associate with class not instance (member type, method, field):	**static**
Class, not instance, initializer (compound-statement outside method):	**static**
Concurrency control (method):	**synchronized**
Forbid extension/modification (class, method, field, variable):	**final**
unused reserved word:	**const**
Non-Java code (method):	**native**
Strictly apply IEEE 754 (class, method):	**strictfp**
Non-persistence (field):	**transient**
Potentially thread-unsafe (field):	**volatile**

Visibility (class): **public** (method, field): **public**, **private**, **protected**

public	potentially visible anywhere	**private**	only in this class
protected	only in subclasses & this package	*default*	only in this package

Constructors

Each constructor starts by calling another constructor, either explicitly by **this (** [arg [**,** arg]*] **);** or **super (** [arg [**,** arg]*] **);** or implicitly by **super ( );**
A class with no explicit constructor gains: **public** classname **( )** { **super ( );** }
Instance initializers and field initialisation code copied into all constructors
Class initializer(s) and static field initialisation code run before class is used

Statements and Control Flow

empty: **;**

declaration: type identifier [= expression] [**,** identifier [= expression]]* **;**

side-effect: expression-with-side-effect **;**

assertion: **assert** boolean-expr [**:** errorcode] **;** *see Assertions*

labelled: label **:** statement

threadsafe: **synchronized (** expression **)** { [statement]* } *see Concurrency*

compound: { [statement]* } *used outside method ⇒ class/instance initializer*

conditionals: **if (** boolean-expr **)** statement [**else** statement]
switch (switch-expr **)** { [**case** value **:** [statement]*]*
[**default :** [statement]*] }
switch-expr has type int, short, char, byte or (in Java 5) equivalent wrapper class or enum type; branches to matching case then falls through cases

loops: **while (** boolean-expr **)** statement
do { [statement]* } **while (** boolean-expr **) ;**
for ([declaration] **;** [boolean-expr] **;** [expr [**,** expr]*] **)** statement
for (uninitialised-variable-declaration **:** iterable-expr **)** statement

jumps: **break** [label] **;** *exits enclosing switch/loop or labelled statement*
continue [label] **;** *skips to next round of enclosing or labelled loop*
NB: **goto** *is an unused reserved word*

invoke: method-expression **(** [arg-expression [**,** arg-expression]*] **)**
invocations are expressions, not statements, but included here for context
overloaded methods match on name & signature wrt actual parameter types

reply: **return** [expression] **;** *value required iff method is non-void*
throw throwable-expression **;** *e.g.* **throw new** throwable-class **(** [args] **);**

handle: **try** { [statement]* }
[**catch (** throwable-type identifier **)** { [statement]* }]*
[**finally** { [statement]* }]
Handle / explicitly propagate Throwables *except* RuntimeException & Error

Expressions & Operators and Strings

Operator	*PA*	*Signature*		*Description*
.	15L	object x name	→ member	member access
[*index***]**	15L	array x int	→ element	array element access
mthd **(***args***)**	15L√	method x args	→ result?	invocation
++ −−	15L√	variable	→ value	post-increment/decrmnt
++ −−	14R√	variable	→ value	pre-increment/decrmnt
∼	14R	integer	→ integer	bitwise complement
!	14R	boolean	→ boolean	boolean NOT
new	13R√	class**(**args**)**	→ object	instance creation
(*type***)**	13R	any	→ any	cast to *type*
∗ / %	12L	number x number	→ number	mult, div, mod
+ −	11L	number x number	→ number	add, subtract
+	11L	string x any	→ string	string concatenation
<<	10L	integer x integer	→ integer	left shift
>>	10L	integer x integer	→ integer	right shift (sign extend)
>>>	10L	integer x integer	→ integer	right shift (zero extend)
< <=	9L	number x number	→ boolean	less than (or equal)
> >=	9L	number x number	→ boolean	greater than (or equal)
instanceof	9L	ref x type	→ boolean	type test
==	8L	builtin x builtin	→ boolean	identical value
!=	8L	builtin x builtin	→ boolean	different value
==	8L	ref x ref	→ boolean	same object
!=	8L	ref x ref	→ boolean	different object
&	7L	integer x integer	→ integer	bitwise AND
&	7L	boolean x boolean	→ boolean	boolean AND
^	6L	integer x integer	→ integer	bitwise XOR
^	6L	boolean x boolean	→ boolean	boolean XOR
|	5L	integer x integer	→ integer	bitwise OR
|	5L	boolean x boolean	→ boolean	boolean OR
&&	4L	boolean x boolean	→ boolean	conditional AND
||	3L	boolean x boolean	→ boolean	conditional OR
? :	2R	boolean x any x any	→ any	ternary if-then-else
=	1R√	variable x any	→ any	assignment
see list below	1R√	variable x any	→ any	operator assignment

operator assignments: ∗= /= %= += −= <<= >>= >>>= &= ^= |=

P = precedence, A = associativity, to override enclose expression in parentheses: **( )**
√ = has side-effect, such expressions can be used as statements by appending a **;**
ref ⇒ object or array reference; variable ⇒ assignable location, e.g. array element

String: (compareTo|equals)[IgnoreCase] contains contentEquals [region]matches trim (ends|starts)With get(Bytes|Chars) [last]indexOf to(Lower|Upper)Case charAt concat split replace[All|First] sub(string|Sequence) to(String|CharArray) length hashcode intern codePoint(At|Before|Count) offsetByCodePoints *statics:* format [copy]valueOf (**Buffer**|**Builder**): append[CodePoint] delete[CharAt] insert replace reverse [last]indexOf

Packages, Jars, Compilation and Execution

Compilation: **javac** [**-classpath** *path*] [**-d** *dir*] [other options]* *file(s)*

Execution: **java** [**-classpath** *path*] [options]* [*package***.**] *classname*

Execution entry point is **public static void main (String [ ]** args **)** in specified class
javac: include **.java** extension; **java**: omit **.class** extension. Classpath lists directories holding package hierarchy roots, **-classpath** overrides CLASSPATH overrides default: **.**
-d directory holds package hierarchy roots for generated classfiles, overrides default: **.**

Packages: name structure: identifier [**.**identifier]* each identifier indicates a directory in a tree-structured hierarchy; trees rooted in classpath dirs; classfiles are placed in, and retrieved from, relevant directory in the tree.

Jar files: like **tar** archives, contain package tree & manifest, held in classpath dirs

On-line documentation and tutorials are available at: http://java.sun.com/
For more detail on Java 5 in printed form, consider *Java in a Nutshell, 5th edition*, produced by David Flanagan, published by O'Reilly, ISBN 0-596-00773-6

Available online at **http://www.dcs.gla.ac.uk/∼pd/JavaRefCard/** *v6.0 r8* (2007/10)

Generics

[**public**] (**interface**|**class**) *name* < [*generic-param* [**,** *generic-param*]*] > { *body* }
simple/wildcard/constrained *generic-param*: (*name* | **?**) [**extends** *type* | **super** *type*]
class generic types used in instance state/methods, not statics; no **new** generic arrays;
static generic methods declare type variable(s) in signature, preceding return type:
i.e. **public static** < *generic-param(s)* > *method-signature* { *method-body* }

Enum Types

javac : enum ⟶ *Comparable Serializable* (non-*Cloneable*) **final** class
values⟶ fixed collection (no public constructor) of **public final static** fields
values can have value-specific methods, these must override enum instance methods
Use **import static** to import all values simultaneously.
Use enums: in switches (cases use value names only, no typename qualifier).
Use enums: as values in Set, List, Map, HashMap, EnumSet; as keys in EnumMap.
No inheritance of/by enums. Only non-public constructors, no explicit **super()** calls.
Additional methods in enum declaration supplement auto-generated methods:

```
public final static E[] values();        public final static E valueOf(String name);
public final String name();              public String toString();
public final int ordinal();              public final int hashCode();
public final int compareTo(E o);         public final boolean equals(Object o);
```

```
[public] enum enum-name [ implements interface-id [, interface-id ]* ] {
    [     NAME [ [(constructor-args)] [ { [method]* } ] ]          // value body
        [, NAME [ [(constructor-args)] [ { [method]* } ] ] ]*      // value body
        [,]                                                        // value list
    [ ; [field]* [initializer]* [constructor]* [method]*  ]        // class body
}
```

Annotations

For tools (execution unaffected) & reflection. Limited retention: SOURCE, CLASS, RUNTIME
Hold named non-null compile-time constants (e.g. annotations) and 1-D arrays of them
At most one annotation of each sort per allowed target; targets are subset of:
TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, ANNOTATION_TYPE, PACKAGE
NB: Local variables and catch clause parameters only accept SOURCE annotations.

@interface creates a new annotation (extending an annotation creates an interface):

```
import java.lang.annotation.*; import static ElementType.*;
@Retention(RetentionPolicy.RUNTIME) @Target({ TYPE, CONSTRUCTOR, METHOD })
public @interface QualityCheck {
    public static enum Quality { BROKEN, POOR, SHIPPABLE };
    String  checkerName();
    Quality quality() default Quality.POOR;
}
```

Each method signature specifies a name-value pair, where value is of return-type
Standard Annotations:
@Deprecated @Override @SuppressWarnings(String[])
Standard Meta-Annotations (annotations with target ANNOTATION_TYPE*):*
@Documented @Inherited @Retention(RetentionPolicy) @Target(ElementType[])
When applying an annotation, can omit: any items for which a default is specified, "value=" if item name is "value", () if no items, and { } for single entry array values

Assertions

assert bool-expr [**:** any]; ⇒ **if** (! bool-expr) **throw new** AssertionError([any]);
assertions are enabled (**–ea**) or disabled (**–da**) at runtime using interpreter flags

–ea	enable assertions in application classes
–ea:package-name**...**	enable assertions throughout specified package
–ea:class-name	enable assertions for specified class
–esa	enable system assertions

long form of flags: **–enableassertions** **–disableassertions** etc

Reflection & Instrumentation

object.**getClass()**, **Class.forName(***classname***)**. *class* can **get**[**Generic**]**Superclass** **get**[**Generic**]**Interfaces** **get**[**Declared**](**Field**|**Constructor**|**Method**)[**s**]. Instantiate with: *class*.**newInstance()**, *constructor*.**newInstance(**[args]**)**. *field* can **get**/**set** value, **getType**. *method* can **get**[**Generic**](**ParameterTypes**|**ExceptionTypes**|**ReturnType**) and is invocable: *method*.**invoke(** Object *obj***,** Object**...** *args***)** ⇒ *obj*.*method*(*args*)
All *members* can **getModifiers** (then test for declaration modifiers & interface-ness), **get**[**Declared**|**Parameter**]**Annotation**[**s**] (RUNTIME retention), **getDeclaringClass**.

A **Proxy** dynamically implements interfaces, delegating invocations to a handler:
newProxyInstance(ClassLoader l**,** Class<?>**[]** interfaces**,** InvocationHandler ih**)**
usage: **if (Proxy.isProxyClass(***object***.getClass()))**
Proxy.getInvocationHandler(*object***).invoke(** *object***,** *method* [**,** *args*]**)**

java –javaagent flag specifies JAR file whose manifest indicates premain class with:
public static void premain(String *args***,** **Instrumentation** *instrument***)**
put ClassFileTransformers into *instrument* to inspect/change bytecodes during loading.

Concurrency Essentials

Simple approach using java.lang.Runnable:

```
public interface Runnable { void run(); }
```

Provide implementation of Runnable objects:

```
public class Foo implements Runnable { public void run() { [statement]* } }
```

Instantiate, and create a thread to execute, a Runnable:

```
Thread t = new Thread(new Foo(args)); t.start();
```

Can specify name and stacksize for Thread. One thread can **interrupt** or **join** another
Current thread can **yield** control, **sleep**, and test whether it **holdsLock** (e.g. in **assert**)
Hierarchically organise/manage threads using java.lang.ThreadGroup

Richer approach uses java.util.concurrent.Callable, Future and ThreadPool Executors

```
package java.util.concurrent;
public interface Callable<V> { V call() throws Exception; }
```

Provide implementation of Callable objects:

```
public class Foo2 implements Callable<Bar2> {
    public Bar2 call() throws Exception { [statement]* }
}
```

Instantiate a Callable and pass it to a ThreadPool, receiving a Future:

```
import java.util.concurrent.*;
ExecutorService pool = Executors.newFixedThreadPool(10);
Future<Bar2> f = pool.submit( new Foo2(args) );
```

Subsequently acquire result from the Future:

```
try { Bar2 b = f.get(); } catch (Exception e) {  }
```

java.util.concurrent.Executors also offers:

newSingleThreadExecutor()	
newCachedThreadPool()	finished threads retained 60 seconds for reuse
newScheduledThreadPool(num**)**	delay/repeat executes Callables/Runnables using **schedule**[**AtFixedRate**|**WithFixedDelay**]

java.util.Timer instances offer **schedule**[**AtFixedRate**] to run **TimerTask** instances
a **java.util.concurrent.DelayQueue** holds **Delayed** (e.g. **ScheduledFuture**) objects

Protect shared objects/state by locking instance/class monitors for critical sections;
threads interact by waiting for/notifying monitors:

```
public class Bar {
    [field-declaration]*
    public Bar(args) { [statement]* }
    synchronized public type methodname(args) { [statement]* }
    [static field-declaration]*
    synchronized public static type methodname(args) { [statement]* }
    public type methodname(args) {
        [statement]*
        synchronized (this) { [statement]* } // Can limit extent of exclusion
        [statement]*
    }
    synchronized public type methodname(args) {
        while ( /* prevented-from-progressing */ )
            try { this.wait(); } catch (InterruptedException e) {}
    }
    synchronized public type methodname(args) {
        this.notifyAll(); // having enabled others to progress
    }
}
```

java.util.concurrent gives additional concurrency control, e.g. instances of:

Semaphore	offer **acquire**[**Uninterruptibly**] **tryAcquire** **release**
locks.ReentrantReadWriteLock	offer **readLock** **writeLock**
locks.Lock	offer **lock**[**Interruptibly**] **tryLock** **unlock** **newCondition**
locks.Condition	offer **signal**[**All**] **await**[**Nanos**|**Until**|**Uninterruptibly**]
CountDownLatch	offer **await** **getNumberWaiting** **reset** **isBroken**
Exchanger	offer **exchange** two threads swap values, re-usable

and **LockSupport** has static **park**[**Nanos**|**Until**] **unpark** to suspend/resume threads
java.util.concurrent.atomic offers atomic operations:

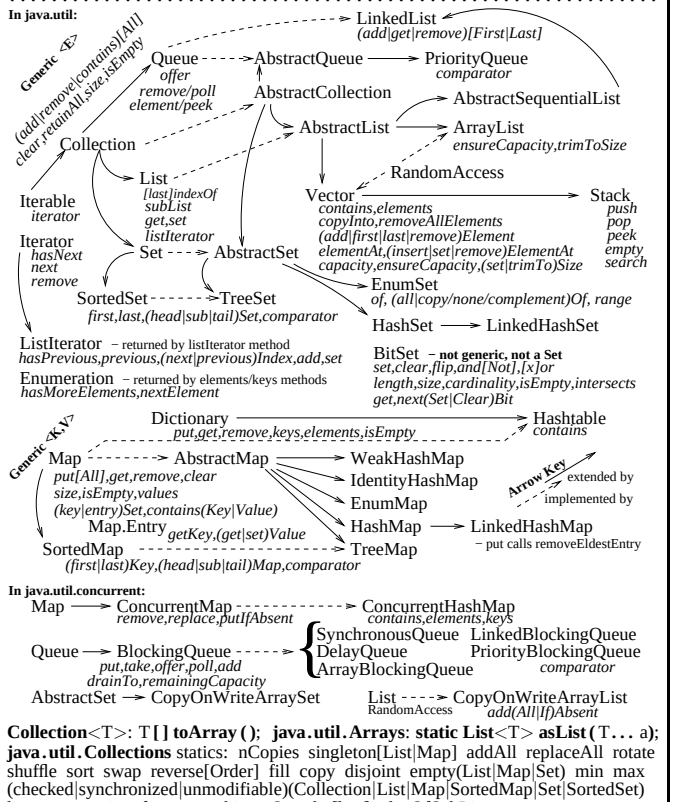
Atomic(Integer|**Long**|**Reference)**[**Array**], **AtomicBoolean** for wrapped values	
Atomic(Integer|**Long**|**Reference)FieldUpdater** on named **volatile** fields all have **set**, **get**[**AndSet**], [**weak**]**compareAndSet**, and the numbers also have (**add**|**decrement**|**increment**)**AndGet**, **getAnd**(**Add**|**Decrement**|**Increment**)	
Atomic[**Markable**|**Stamped**]**Reference** combine a boolean or int with a reference both offer **set**, **getReference**, [**weak**]**compareAndSet** and **isMarked**, **attemptMark** or **getStamp**, **attemptStamp** respectively	

Use **java.lang.**[**Inheritable**]**ThreadLocal**<T> to **set**, **get**, **remove** per-thread values

Nested Types / Inner Classes

static member types (class, interface, enum, annotation): nested in top-level types or static member types, named by concatenating enclosing name(s) with **.** separator
non-static member classes: one instance per instance of enclosing class/enum;
no static content (except constants); separate containment/inheritance hierarchies;
if extended by non-contained class, must provide an "enclosing instance" to constructor
local classes: declared in compound statement; access instance fields & methods, final local variables, method & exception parameters; closure-like; lexical ≠ temporal scope
anonymous classes: single-instance, un-named, no-constructor local class
syntax: **new** class/interface-name { body extends class/implements interface }
Reflection support for nested types includes methods in **Class**:
getEnclosing(Class|**Method**|**Constructor)** **is(Member**|**Local**|**Anonymous)Class**

Collection<E> Map<K,V> & BitSet Essentials



Collection<T>: T**[]** **toArray()**; **java.util.Arrays**: **static List**<T> **asList(**T**...** a);
java.util.Collections statics: nCopies singleton[List|Map] addAll replaceAll rotate shuffle sort swap reverse[Order] fill copy disjoint empty(List|Map|Set) min max (checked|synchronized|unmodifiable)(Collection|List|Map|SortedMap|Set|SortedSet)
list enumeration frequency binarySearch [last]indexOfSubList EMPTY_(LIST|MAP|SET)

Simple Text I/O Essentials

Output to a **java.io.PrintStream** using: **print** **println** **append** **format** **printf**
Example targets: **System.out** **System.err** **new PrintStream(new File(***pathname***))**
java.util.Formatter can **format** to any **Appendable**, e.g. **File** **String** **PrintStream**
Enriched C-style formats: %% %n %[arg][flags][width][.precision] type
arg: < reuse previous, *n*$ use arg n *flags:* −#+ **(**0, *type:* cs b dox efga t?
java.util.Scanner reads **Readable**, e.g. **File** **String** **InputStream** (e.g. **System.in**), by [**has**] **next**[**Line**|**Boolean**|**Double**|**Float**|**Byte**|**Int**|**Short**|**Long**|**BigInteger**|**BigDecimal**]

Reference Sheet Notation: [ ] ⇒ optional; [ ]* ⇒ ≥ 0; [ ]+ ⇒ ≥ 1; (|) ⇒ choice
Related Java Notation: **[ ]** ⇒ array index/decl; ∗ ⇒ asterisk; + ⇒ plus; | ⇒ or

Available online at **http://www.dcs.gla.ac.uk/~pd/JavaRefCard/** *v6.0 r8* (2007/10)