Java Language Quick-Reference Guide

Console Output

Java applications and applets can output simple messages to the console as follows:

```
System.out.println("This is displayed on the console");
```

Data Types

`boolean`	Boolean type, can be `true` or `false`
`byte`	1-byte signed integer
`char`	Unicode character (i.e. 16 bits)
`short`	2-byte signed integer
`int`	4-byte signed integer
`long`	8-byte signed integer
`float`	Single-precision fraction, 6 significant figures
`double`	Double-precision fraction, 15 significant figures

Operators

+ - * / %	Arithmetic operators (% means *remainder*)
++ --	Increment or decrement by 1 result = ++i; means increment by 1 first result = i++; means do the assignment first
+= -= *= /= %= etc.	E.g. i += 2 is equivalent to i = i + 2
&&	Logical AND, e.g. if (i > 50 && i < 70) The second test is only carried out if necessary - use & if the second test should *always* be done
||	Logical OR, e.g. if (i < 0 || i > 100) The second test is only carried out if necessary - use | if the second test should *always* be done
!	Logical NOT, e.g.if (!endOfFile)
== != > >= < <=	Relational operators
& | ^ ~	Bitwise operators (AND, OR, XOR, NOT)
<< >> >>>	Bitwise shift operators (shift left, shift right with sign extension, shift right with 0 fill)
instanceof	Test if an object is an instance of a class, e.g. if (anObj instanceof BankAccount) System.out.println("$$$");

Control Flow—if … else

`if` statements are formed as follows (the `else` clause is optional). The braces {} are necessary if the if-body exceeds one line; even if the if-body is just one line, the braces {} are worth having to aid readability:

```
String dayname;
…
if (dayname.equals("Sat") || dayname.equals("Sun")) {
   System.out.println("Hooray for the weekend");
}
else if (dayname.equals("Mon")) {
   System.out.println("I don't like Mondays");
}
else {
   System.out.println("Not long for the weekend!");
}
```

Control Flow—switch

`switch` is used to check an integer (or character) against a fixed list of alternative values:

```
int daynum;
…
switch (daynum) {
   case 0:
   case 6:
      System.out.println("Hooray for the weekend");
      break;

   case 1:
      System.out.println("I don't like Mondays");
      break;

   default:
      System.out.println("Not long for the weekend!");
      break;
}
```

Control Flow—Loops

Java contains three loop mechanisms:

```
int i = 0;
while (i < 100) {
   System.out.println("Next square is: " +  i*i);
   i++;
}
```

```
for (int i = 0; i < 100; i++) {
   System.out.println("Next square is: " +  i*i);
}
```

```
int positiveValue;
do {
   positiveValue = getNumFromUser();
}
while (positiveValue < 0);
```

Defining Classes

When you define a class, you define the data attributes (usually `private`) and the methods (usually `public`) for a new data type. The class definition is placed in a `.java` file as follows:

```
// This file is Student.java. The class is declared
// public, so that it can be used anywhere in the program

public class Student {

   private String name;
   private int    numCourses = 0;

   // Constructor to initialize all the data members
   public Student(String n, int c) {
      name = n;
      numCourses = c;
   }

   // No-arg constructor, to initialize with defaults
   public Student() {
     this("Anon", 0);       // Call other constructor
   }

   // finalize() is called when obj is garbage collected
   public void finalize() {
     System.out.println("Goodbye to this object");
   }

   // Other methods
   public void attendCourse() {
     numCourses++;
   }

   public void cancelPlaceOnCourse() {
     numCourses--;
   }

   public boolean isEligibleForChampagne() {
     return (numCourses >= 3);
   }
}
```

Using Classes

To create an object and send messages to the object:

```
public class MyTestClass {

   public static void main(String[] args) {

      // Step 1 - Declare object references
      // These refer to null initially in this example
      Student me, you;


      // Step 2 - Create new Student objects
      me  = new Student("Andy", 0);
      you = new Student();


      // Step 3 - Use the Student objects
      me.attendCourse();
      you.attendCourse();


      if (me.isEligibleForChampagne())
         System.out.println("Thanks very much");

   }
}
```

Arrays

An array behaves like an object. Arrays are created and manipulated as follows:

```
// Step 1 - Declare a reference to an array
int[] squares;          // Could write int squares[];


// Step 2 - Create the array "object" itself
squares = new int[5];  // Creates array with 5 slots


// Step 3 - Initialize slots in the array
for (int i=0; i < squares.length; i++) {
   squares[i] = i * i;
   System.out.println(squares[i]);
}
```

Note that array elements start at [0], and that arrays have a length property that gives you the size of the array. If you inadvertently exceed an arrays' bounds, an exception is thrown at run time and the program aborts.

Note: Arrays can also be set up using the following abbreviated syntax:

```
String[] cities = {
     "San Francisco",
     "Dallas",
     "Minneapolis",
     "New York",
     "Washington, D.C."
};
```

Inheritance and Polymorphism

A class can inherit all of the data and methods from another class. Methods in the *superclass* can be over-ridden by the subclass. Any members of the superclass that you want to access in the subclass should be declared `protected`. The `protected` access specifier allows subclasses, plus any classes in the same package, to access that item.

```
public class Account {
   private double balance = 0.0;

   public Account(double initBal) {
      balance = initBal;
   }

   public void deposit(double amt) {
      balance += amt;
   }

   public void withdraw(double amt) {
      balance -= amt;
   }

   public void display() {
      System.out.println("Balance is: " + balance);
   }
}

public class CheckAccount extends Account {
   private int maxChecks = 0;
   private int numChecksWritten = 0;

   public CheckAccount(double initBal, int maxChk) {
      super(initBal);            // Call superclass ctor
      maxChecks = maxChk;        // Initialize our data
   }

   public void withdraw(double amt) {
      super.withdraw(amt);       // Call superclass
      numChecksWritten++;        // Increment chk. num.
   }

   public void display() {
      super.display();           // Call superclass
      System.out.println(numChecksWritten);
   }
}
```

Abstract Classes

An abstract class is one that can never be instantiated; in other words, you cannot create an object of such a class. Abstract classes are specified as follows:

created and

```
// Abstract superclass
public abstract class Mammal {
   …
}


// Concrete subclasses
public class Cat extends Mammal {
   …
}

public class Dog extends Mammal {
   …
}

public class Mouse extends Mammal {
   …
}
```

Abstract Methods

An abstract method is one that does not have a body in the superclass. Each concrete subclass is obliged to override the abstract method and provide an implementation; otherwise, the subclass is itself deemed abstract because it does not implement all its methods.

```
// Abstract superclass
public abstract class Mammal {

   // Declare some abstract methods
   public abstract void eat();
   public abstract void move();
   public abstract void reproduce();

   // Define some data members if you like
   private double weight;
   private int age;

   // Define some concrete methods too if you like
   public double getWeight{} {
      return weight;
   }

   public int getAge() {
      return age;
   }
}
```

Interfaces

An interface is similar to an abstract class with 100% abstract methods and no instance variables. An interface is defined as follows:

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

A class can implement an interface as follows. The class is obliged to provide an implementation for every method specified in the interface, otherwise the clas must be declared `abstract` because it doesn't implement all its methods..

```
public class MyApp extends Applet implements Runnable {

   public void run() {
      // This is called when the Applet is kicked off
      // in a separate thread
      …
   }


   // Plus other applet methods
   …
}
```

Static Variables

A `static` variable is like a global variable for a class. In other words, you only get one instance of the variable for the whole class, regardless of how many objects exist. `static` variables are declared in the class as follows:

```
public class Account {
   private String accnum;               // Instance var
   private double balance = 0.0;        // Instance var

   private static double intRate = 5.0;  // Class var

   …
}
```

Static Methods

A `static` method in a class is one that can only access `static` items; it cannot access any non-`static` data or methods. `static` methods are defined in the class as follows:

```
public class Account {

   public static void setIntRate(double newRate) {
      intRate = newRate;
   }

   public static double getIntRate() {
      return intRate;
   }
   …
}
```

To invoke a `static` method, use the name of the class as follows:

```
public class MyTestClass {

   public static void main(String[] args) {
      System.out.println("Interest rate is" +
                         Account.getIntRate());
   }
}
```

Packages

Related classes can be placed in a common package as follows:

```
// Car.java
package mycarpkg;

public class Car {
   …
}


// Engine.java
package mycarpkg;

public class Engine {
   …
}


// Transmission.java
package mycarpkg;

public class Transmission {
   …
}
```

Importing Packages

Anyone needing to use the classes in this package can *import* all or some of the classes in the package as follows:

```
import mycarpkg.*;      // import all classes in package
```

or

```
import mycarpkg.Car;    // just import individual classes
```

The `final` Keyword

The `final` keyword can be used in three situations:

`final` classes (for example, the class cannot be inherited from)
`final` methods (for example, the method cannot be overridden in a subclass)
`final` variables (for example, the variable is constant and cannot be changed)

Here are some examples:

```
// final classes
public final class Color {
   …
}


// final methods
public class MySecurityClass {
   public final void validatePassword(String password) {
      …
   }
}


// final variables
public class MyTrigClass {
   public static final double PI = 3.1415;
   …
}
```

Exception Handling

Exception handling is achieved through five keywords in Java:

`try`	Statements that could cause an exception are placed in a 'try' block
`catch`	The block of code where error processing is placed
`finally`	An optional block of code after a 'try' block, for unconditional execution
`throw`	Used in the low-level code to generate, or 'throw' an exception
`throws`	Specifies the list of exceptions a method may throw

Here are some examples:

```
   public class MyClass {

         public void anyMethod() {
                  try {
                     func1();
                     func2();
                     func3();
                  }
                  catch (IOException e) {
                     System.out.println("IOException:" + e);
                  }
                  catch (MalformedURLException e) {

System.out.println("MalformedURLException:" + e);
                  }
                  finally {
System.out.println("This is always displayed");
                  }
         }

         public void func1() throws IOException {
                  …
         }

         public void func2() throws MalformedURLException {
                  …
         }

         public void func3() throws IOException,
                                    MalformedURLException {
                  …
         }
   }
```