

Java Review Sheet

Declaring Classes

```
[public] [abstract] [final] class ClassName [extends Super] [implements Interface] {
    // field and method declarations
}
```

public: class can be used by any other class

abstract: class cannot be instantiated

final: class cannot be subclassed

class: declares class with *ClassName*

extends: identifies *Super* as the superclass of the class

implements: identifies *Interface* as used by the class

Declaring Interfaces

```
[public] interface InterfaceName [extends Super] {
    // no data or bodies; implementing class must use all methods
}
```

Defining Methods

```
[public/protected/private/default] [static] [abstract] [final] [native]
[synchronized] void/returnType main/methodName(paramList)* [throws
exceptionList] {
    // method body; a return [value]; statement exits the method
    immediately
}
```

	Class	Package	Subclass	World
private	Y	N	N	N
default	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y

static: declares as a class method rather than an instance method

abstract: is a member of an abstract class

final: cannot be overridden by subclasses

native: used to implement libraries not written in Java

synchronized: limits access to one thread at a time

void:

main: identifies method that command line arguments may be passed to, takes "String[] args" in *paramList*

(*paramList*): arguments for the method; *dataType* first, *variableName* second

throws: throws checked exceptions in *exceptionList*

return: terminates the current method; return a value to the method's

caller by using **return value**;

* methods may be "overloaded" if two methods have the same *methodName* but different (*paramList*)

Constructors

```
ClassName(paramList) {
    // constructor body; cannot return values, invoked with new object
}
```

Instance Variables

```
class ClassName {
    [public/protected/private] [static] [final] dataType variableName
    [= value];
}
```

public: anyone can access

protected: only methods of the same package or of its subclasses can access

private: only methods of the same class can access

static: variable is associated with the class, not an instance of the class

final: variable value cannot be altered

Enum Types

```
class ClassName {
    [public/protected/private] enum name {VALUE_NAME0,
        VALUE_NAME1, ... , VALUE_NAMEn-1}
    name vi = name.VALUE_NAMEi;
    name wi = name.valueOf("VALUE_NAMEi");
    // methods, etc. . .
}
```

Arrays

```
class ClassName {
    [public/protected/private] [static] [final] dataType[] arrayName;
}
```

dataType: number arrays are initiated with all 0's, object arrays are initiated with all **null**'s; arrays are indexed using the integer set {0, 1, . . . , *arrayName.length-1*} and are created with **new dataType[length]**;

Local Variables

```
modifierType returnType methodName (paramList) {
    dataType variableName1; // do not exist outside methodName
    dataType variableName2 = initialValue;
```

```
}
```

Creating and Using Objects

create: **new** *ClassName* (*argList*)

method: *objectReference.methodName*(*argList*)

instance variable: *objectReference.variableName*

Base Types

boolean: Boolean value (*true* or *false*)

char: 16-bit Unicode character (*1-9, A-Z, a-z, etc. . .*)

byte: 8-bit two's complement integer (-128 to 127)

short: 16-bit two's complement integer (-32768 to 32767)

int: 32-bit two's complement integer (-2147483638 to 2147483647)

long: 64-bit two's complement integer (-9223372036854775808 to 9223372036854775807)

float: 32-bit floating-point number (-3.4028236E38 to 3.4028235E38)

double: 64-bit floating-point number (-1.7976931348623158E308 to 1.7976931348623157E308)

Number and String Objects

	Class Name	Creation	Access
byte	Byte	n = new Byte(<i>(byte)value</i>)	n.byteValue()
short	Short	n = new Short(<i>(short)value</i>)	n.shortValue()
int	Integer	n = new Integer(<i>value</i>)	n.intValue()
long	Long	n = new Long(<i>valueL</i>)	n.longValue()
float	Float	n = new Float(<i>valueF</i>)	n.floatValue()
double	Double	n = new Double(<i>value</i>)	n.doubleValue()
string	String	String s = " <i>value</i> "	*

* examples include s.length and others

Operators

assignment (=): evaluated "left to right"; i = j = 25

arithmetic (+, -, /, *, %): follows order of operations

increment (n++, ++n)

decrement (n--, --n)

logic (<, <=, ==, !=, >=, >): for comparisons that yield **boolean** types

not/and/or (!, &&, ||): operate on **boolean** values

concatenation (+): "string1" + "string2" -> "string1string2"

Changing Data Type/Casting

numberDataTypeA variableNameA = *value*;

numberDataTypeB variableNameB = (*numberDataTypeB*)*value*; // explicit

doubleVariableB = *intVariable* / *doubleVariableA*; // implicit

intVariableB = *intVariableA* / *doubleVariable*; // implicit; gives compile error

String *stringName1* = Integer.toString(*intValue*); //explicit

String *stringName2* = "" + *intValue*; // implicit

Looping Statements

```
while (exprA comparisonOp expB or method that gives boolean) {
    // instructions to be executed if true
}
```

```
do {
    // statements to be executed
} while (exprA comparisonOp expB or method that gives boolean);
```

```
for (initialization; comparison or method that gives boolean; incrementer) {
    // instructions to be executed if true
}
```

initialization: **int count** = 0 or some such statement

increment: *count* + *increment* or some such statement

Decision-Making Statements

```
if (comparison or method that gives boolean) {
    // statements to be executed if comparison is true
}
```

```
else if (2nd comparison or method that gives boolean) {
    //statements to be executed if 2nd comparison is true and 1st is false
}
```

```

...
else if (nth comparison or method that gives boolean) {
    // statements to be executed if nth comparison is true and the
    // previous ones are false
}
else {
    // statements to be executed if all comparisons above are false
}

switch (integerTypeVariable or enumTypeVariable) {
    case integerValueA/enumValueA:
        // statements for case A
        // insert break [label]; here to leave switch, otherwise
        // next case evaluated
    case integerValueB/enumValueB:
        // statements for case B
    ...
    default:
        // statements if none of the above apply
    break [label]/continue/return [value];
}
break [label]: a statement is labeled statementName: for/while/switch/do;
the unlabeled form of break terminates the innermost
switch/for/while/do and the labeled form of the break statement
terminates to an outer switch/for/while/do with the given label.
continue: terminates the current iteration of the innermost loop and evaluates
the boolean expression that controls the loop; the labeled form
skips the current iteration of the loop with the given label.
return: terminates the current method; return a value to the method's
caller by using return value;

```

Exception-Handling Statements

throw new *exceptionName(exceptionParameters)*
exceptionParameters: constructor values for the *exceptionName* object

```

try {
    // main block of statements to check for thrown exceptions
}
catch (exceptionNameA exceptionVarA)
{
    // exceptionNameA handling statements that use exceptionVarA
}
catch (exceptionNameB exceptionVariableB) {
    // exceptionNameB handling statements that use exceptionVarB
}
...
finally {
    // optional statements executed before exiting try-catch block
}

```

Simple Input/Output

print(Object *objectName*/String *stringName*/*dataType value*); // prints
objectName or *stringName* or *value*

println(String *stringName*); // prints the string *stringName*, followed by \n

```

import java.io.*; // for a scanner class
import java.util.Scanner;
[public] class InputExample {
    public static void main(String args[]) throws IOException {
        Scanner s = new Scanner(System.in);
        System.out.print(promptString);
        float info = s.nextFloat();
    }
}

```

Scanner Methods

.hasNext(): returns **true** iff there is another token in the input stream
.next(): return the next token stream in the input stream
.hasNextType(): returns **true** iff there is a token with data type *Type*
.nextType(): returns the next token as *Type*

Java Style

braces: always use braces – even when not required; line up braces
whitespace: use vertical whitespace to organize the body of your code into
readable parts; use horizontal whitespace to organize each line of
your code into readable parts; use tabs and spaces to indicate
nesting

indentation: lines must not exceed 80 columns in length including whitespace;
use tabs and spaces to indicate nesting

naming: source files are named the same as their class with a .java extension;

variable, method, and package names begins with a lower case
letter while class names begin with an upper case letter;
CONSTANTS are in all upper case letters;
nameWithMultipleWords use initial capitals to make subsequent
words distinct (CONSTANTS use underscores); use descriptive,
relevant names; 1 letter names can be used for temporary
variables and loop counters

Commenting

file headers: the source file for the main class must have a main file header
comment

class headers: each class must have a header comment before the class
declaration

method headers: each method must have a header comment before the method
declaration

variable declarations: variables must be commented detailing their use;
primitive variable declarations must also specify their range of
values

other comments:

- highlight major steps of an algorithm
- explain long calculations or conditions
- clarify convoluted or unusual code
- mark locations where you suspect a bug may exist
- mark locations where improvements or enhancements are
planned