CS 367 Java Syntax Review Sheet

Simple Stuff

Hello World Program

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

Variable Definition

typeName variableName = value; // Assignment optional
int thisInt = 0;

Variable Assignment

variableName = value;
thisInt = 10;

Object Construction

ClassName = new *ClassName*(*parameters*)
thisClass = new thisClass();

Importing a Class from a Package

import *packageName.ClassName*;
import java.swing.*;

Outputting to the console

System.out.print(*parameters*);
System.out.println(*parameters*); // Outputs *parameters* and new line character

Classes

Method Definition

accessSpecifier returnType methodName(*parameterType parameterName*)
{
    Method body
}

```
public void thisMethod()
{
		doSomething();
}
```

Constructor Definition

accessSpecifier ClassName(*parameterType parameterName*)
{
    Constructor body
}

```
public thisClass()
{
		doSomething();
}
```

Instance Field Declaration

accessSpecifier fieldType fieldname;

The return Statement

return *expression*; // Expression optional
This statement is to be used when you want the program to break out of a method. But it can also be used to break out of loops, and if statements.

Data Types

Type	Description	Size
int	integers, range : positive and negative 2 billion	4 bytes
byte	single byte, range: -128 to 127	1 byte
short	Short integer, range: -32768 to 32767	2 bytes
long	Long integer, range: positive and negative 9E18	8 bytes
double	Double precision, 15 significant digits	8 bytes
float	Single precision, 7 significant digits	4 bytes
char	Character, Unicode encoding scheme	2 bytes
boolean	True or false	1 bit

Casting

(*typeName*)*expression*

Constant Definition (Final)

In method: final *typeName variableName = expression*;
In class: *accessSpecifier* static final *typeName variableName = expression*;

Static Method Call

ClassName.methodName(*parameters*)

Decisions

The if Statement

if (*condition*)
{ *Statements* }
else if(*condition*)
{ *Statments*}
else
{ *Statements* }

```
if(x==y)
{
		doSomething();
}
else if (x==z)
{
		doSomethingElse();
}
else
{
		doThis();
}
```

The Selection Operator

condition ? *value1* : *value2* so if *condition* is true it returns *value1*

The switch Statement (example)

```
int digit;
…
switch (digit)
{
		case 1: … break;
		case 2: … break;
		…
		default: … ;
}
```

Enumerated Types, Definition

accessSpecifier enum *TypeName* { *value1*, *value2*, … }
public enum Game {Win, Loss, Draw};
Enumerators is a set of constants represented by identifiers.

Iteration (Loops)

while Loop

while (*condition*)
{ *statements* }

```
while(x==y)
{
		doSomething();
}
```

do Loop

do
{ *statements* } while (*condition*);

```
do
{
		doSomething();
}while(x==y);
```

The difference between a while and a do while loop is that a do while loop will at lease run once.

for Loop

for (*initialization*; *condition*; *increment*)
{ *statements* }

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

break and continue statements

break; // breaks the current loop
continue; // goes to the bottom of the current loop
break *label*; // breaks out of the loop labeled with *label*

```
loop1:for(i=0;i<10;i++)
{
		while(x==y)
		{
		  doSomething();
		  break loop1;
		  }
}
```

Random number generator, Definition and use

Random rand = new Random();
rand.nextInt(*n*); // returns a random integer between 0 and n-1 (inclusive)
rand.nextDouble(); // returns a random double between 0 and 1, but not 1

Arrays

Array Construction
new *typneName*[*length*]
thisArray[] = new thisArray[x];
Array Element Access
arrayReference[*index*]
thisArray[0];
Array Initialization
type[ ] *reference* = new *type*[*length*];
thisArray[] = new thisArray[x];
type[ ] *reference* = { *value1*, *value2*, ... };
thisArray[] = {a, b, c, d};

Interfaces

Defining an Interface
public interface *InterfaceName*
{ *method signatures* }
An interface is used as a template for any class that implements it. Forcing that class to declare certain methods and variables.

Implementing an Interface
Public class C*lassName* implements *InterfaceName*
{ *interface methods* }
An Interface will not contain any real information.

Inheritance

Extending a Class
class *SubclassName* extends *SuperClassName*
{ *methods and fields* }
A class that is extending another inherits methods and variables from the extended class.

Calling a Superclass Method
super.*methodName*(*parameters*);

Calling a Superclass Constructor
ClassName(*parameters*)
{ super(*parameters*); … }

The instanceof Operator
object instanceof *TypeName* //returns true if *object* is an instance of *TypeName*

Overloading
A class that is extending another class may declare the same methods and variables that the extended class is giving it. By doing this when the extending class calls those methods and variables it will use it own. While if the extended class calls those methods and variables it will also use it's own.

Exceptions

Throwing an Exception
throw *exceptionObject*; // throw new IllegalArgumentException();
throw thisException;

Exception Specification
accessSpecifier returnType methodName(*parameters*) throws *Exception*
public void thisMethod() throws thisException
{}

Try-Catch Block
try { *statements* }
Catch (*ExceptionClass ExceptionObject*) { *statements* }

```
Try
{
            doSomething();
}catch(thisException te)
{
            doSomethingElse();
}finally
{
            doThis();
}
```

UML Class Diagrams

Class Name
visibility dataMemberReference : *type* …
visibility methodReference(*parameter types*) : *returnType*

Generics
Generics are special classes with no specific type definitions.

Ex:

```
public class Pair<K, V> {
  K key;
  V value;
  public void set(K k, V v){
    key = k;
    value = v;
  }
  public K getKey() {return  key;}
  public V getValue() { return  value;}
  public String toString() {
    return "[" + getKey() + ", " + getValue() + "]";
  }
  public static void main(String[] args) {
    Pair<String, Integer> pair1 = new Pair<String,Integer>();
    Pair1.set(new String("height"),  new  Integer(36));
    System.out.println(pair1);
  }
}
```

When you create an instance of a Generic like this, you must define the data types in the declaration line:
Pair<String, Integer> pair1 = ***new*** *Pair<String,Integer>();*
Since this is a **very** generic class definition you can use any class you wish:
Pair<Double, Float> pair1 = ***new*** *Pair<Double, Float>();*

** If you save this class the file name would be: ***Pair.java*** **

Scanner Class
The new Scanner class makes it easier to read information from the console.
** To use this feature you must **import java.util.Scanner**

Declare a scanner object:
Scanner s = new Scanner(System.in);

Using Scanner (a simple example):
Scanner s = new Scanner(System.in);
System.out.print("Enter your age: ");
int age = s.nextInt();
System.out.println("Your age is: " + age);

Useful Scanner Methods:
hasNext(): returns true if and only if there is another token in the input stream

next(): returns the next token in the input stream; it will generate an error if there are no more tokens

hasNext*Type*(): returns true if there is another token of type ***Type***. Where ***Type*** is Boolean, Byte, Double, Float, Int, Long, or Short.

next*Type*(): returns the next token of type ***Type.*** Where ***Type*** is Boolean, Byte, Double, Float, Int, Long, or Short.

hasNextLine(): returns **true** if and only if there is another line of text in the input stream.

nextLine(): Advances the input past the current line ending and returns the input that was skipped. *Basically reads in the current line of text and advanced to the next line.*

findInLine(String s): Attempts to find a string matching the pattern *s*. If the pattern is found, it is returned and the scanner advances to the first character after the match. If it is not found it returns null and doesn't advance.