Terminology for Java Classes

Javadoc comment

Class definition

Constant

Field declarations

Visibility modifiers

Constructor

Parameter (cost)

Accessor methods (getters)

Return type (int)

Mutator methods (setters)

Method signature

Method body

```
/* TicketMachine models a naive ticket machine
 * @author David J. Barnes and Michael Kölling
 * @version 2011.07.31
 */

public class TicketMachine
{
    public final static int MAX_PRICE = 100;

    private int price;
    private int balance;

    /**
     * Create a machine that issues tickets
     */
    public TicketMachine(int cost)
    {
        price = cost;
        balance = 0;
    }



    public int getPrice() { return price; }

    public int getBalance() { return balance; }



    public void insertMoney(int amount)
    {
        balance = balance + amount;
    }

    public void printTicket()
    {
        System.out.println("# Ticket " +
             price + " cents.");
        balance = 0;
    }
}
```

Java Data Types	Declare and Instantiate	Examples of Use

boolean, int, double, char
 	
byte, short, long, float
	boolean found = false;
	int i;
	double pi = 3.14;
	char ch = 'a';
		(!found && i<10)
		i++;
		i=i+1;
		a = r*r*pi;

String	String msg1 = "hello ";
	String msg2 = "there"
		msg1+msg2; //"hello there"
		msg1.equals("hello ");
		msg1.contains("lo");

ArrayList<E>	ArrayList<String> names = new ArrayList<String>();
	
	ArrayList<Student> classlist;
		names.add("Ben");
		names.add("Moose Smith");
		names.get(names.size()-1);
		//get last name
		classlist = new ArrayList<Student>();
		classlist.add(new Student("fred",100));

arrays	int[] mode = new int[10];
	
	String[] = new String[5];
		int places = new int[] {1,2,3,4,5};
		(places.length == 5)
		places[0]=places[0] - 2;
		//was 1 now -1
		places[4]=places[3] * 3;
		//pos 4 w 5 now 12

HashMap<K,V>	HashMap<String,String> phonebook;
	
	phonebook = new HashMap<String,String>();
		phonebook.put("fred","+61 345 6677");
		phonebook.get("fred");
		//is String "+61…"
		phonebook.size()
		phonebook.keySet()
		//all the keys (names)
		phonebook.values()
		//all the values (ph nums)

HashSet<E>	HashSet<String> words = new HashSet<String>();	for (String word : wordArray) {
		  words.add(word);
		}
		words.contains("hello")
		words.size()

Programming Patterns

Programming patterns correspond to fragments of code that accomplish common programming goals. Bergin notes that "A "pattern" is a solution to a problem in a context. … a pattern is an attempt to establish "best practice" with respect to a problem or class of problems."

Whether Or Not

Whether Or Not is a selection pattern. Bergen describes the Whether Or Not pattern as:

``You are in a situation in which some action may be appropriate or inappropriate depending on some testable condition. ... You don't need to repeat the action, only to decide whether or not it should be done. There are no other actions to do instead of this one. You want to write simply understood code."

For example: a bank deposit method should only update the balance if the amount deposited is positive:

```
if (amount > balance) {
     balance  = balance + amount;
}
```

Whether Or Not with warning

Barnes and Kölling use a variant of the Whether Or Not pattern in which an else branch containing only print statement(s) is used to inform the user of the error condition. For example,

```
if (amount > balance) {
    balance  += amount ;
} else {
    System.out.println("amount must be >0");
}
```

Alternative Action (Selection)

Bergen describes this pattern as follows:

You are in a situation in which one of exactly two actions is appropriate depending on some testable condition.

When the condition holds you want to do one action, and when it does not hold you want to do some different action. There are exactly two actions and exactly one condition, which may be true or false.

Therefore, use a single IF statement with an ELSE part, expressing the test as a positive condition.

IF <condition>
    <one action>
ELSE
    <another action>

For example a student may pass or fail an exam depending on the value of the numeric grade.

```
if ( numericGrade > 50 ){
    output ("passing");
} else {
    output("failing");
}
```

Process All Items

Process All Items is a loop pattern. Astrachan and Wallingford describe the Process All Items pattern as: ``The items are stored in an array a. Use a definite loop to process all the items." For example,

```
for(int k=0; k < a.length; k++) {
  process a[k ];
}
```

Barnes and Kölling give a for-each version of this pattern for collections (and arrays since Java 5)

```
ArrayList<String> names = new ArrayList<String>();

for  (String ni: names) {
     process ni;
}
```

Search

You are working with a collection or stream of objects.

How do you find an object that meets a specific condition?

Suppose that you have a set of students, and you would like to find the first student with an "A" average. In the worst case, you will look at the whole collection before finding your target. But, if you find a match sooner, you would like to terminate the search and work with the object that you found.

Therefore, construct a Process All Items loop, but provide a way to exit the loop early should you find your target sooner.

```
ArrayList<String> names = new ArrayList<String>();
String key; //String to search for

for  (String ni: names) {
     if (ni.equals(key)) {
            return true; //key is found
     }
}
return false; //when key not found
```

Temporary variable patterns

The process all items pattern usually requires some local variable(s).
Beck describes several different patterns for local variable use:
``How do you save the value of an expression for later use within a method? ... wherever possible create variables whose scope and extent is a single method. Declare them just below the method selector. Assign them a value as soon as the expression is valid. ...

Collecting Temporary Variable ... saves intermediate results for later use.

```
int[]a = new int[] {0,-1,3,4,-5,-6,10};
int countnegs = 0;
for  ( int ai : a ) {
     if (ai < 0) {
            countneg++;
     }
}
return countneg;
```

Caching Temporary Variable ... improves performance by saving values.

Explaining Temporary Variable ... improves readability by breaking up complex expressions."

```
double[] scores = new double[12];
double avg = average(scores);
double sumsq = 0;
for  ( double si : scores ) {
     double diff =  (ai – avg) * (ai – avg);
     sumsq = sumsq + diff;
}
double stddev = (sum / a.length);
return stddev;
```

In the example above, `stddev` and `diff` are explaining temporary variables. `sumsq` is s collecting variable and `avg` is a caching temporary variable.

The Collecting temporary variable pattern is often combined with the Process All Items pattern to evaluate extreme values and for summing or counting a collection.

Further Reading

O. Astrachan and E. Wallingford. Loop patterns: Definite process all items, 1998. Retrieved March 2012 from `http://www.cs.duke.edu/~ola/patterns/plopd/loops.html`

D. J. Barnes and M. Kolling. Objects First with Java: A Practical Introduction using BlueJ. Prentice Hall, Pearson Education, 5th edition, 2012.

K. Beck. Portland pattern repository: Temporary variables, 1995. Retrieved March 2012 from `http://c2.com/ppr/temps.html`

J. Bergin. Patterns for Selection Version 4, 1999. Retrieved March 2009 from `http://csis.pace.edu/~bergin/patterns/Patternsv4.html`

Properties of Java Applications

Most of the definitions in this section are from Barnes and Kölling, Objects First With Java.

Syntax

The syntax of the Java programming language is the set of rules that defines how a Java program is written and interpreted.
Java's grammar rules define the legal forms of Java statements, including correct use of brackets, semi-colons and key words.
Java's static typing rules check that all expressions are correctly typed.
The Java compiler flags syntax errors and type errors in Java code.

Logical Correctness

A logical error is a problem where the program compiles and executes, but delivers the wrong result. Techniques such as code inspection and JUnit unit testing can be used to help to identify logical errors.

Style and Readability

A major goal of any software developer should be to write consistently clear, high quality, maintainable code. This is not always easy and requires a certain amount of discipline at the best of times. One way to help achieve high quality code is via the use of coding standards. The programming style guide used in CITS1001 is available from http://www.csse.uwa.edu.au/UWAJavaTools/checkstyle/ and is based on the Barnes and Kölling Objects First with Java - Style Guide Version 2.0 from www.bluej.org/objects-first/styleguide.html (and see appendix)

Coupling

Coupling describes the interconnectedness of classes. We strive for **loose coupling** in a system – that is, a system where each class is largely independent and communicates with other classes via a small, well-defined interface.
Well-designed classes should hide implementation information from view, making only information about *what* a class can do should be visible to the outside, but not *how* it does it. Proper encapsulation in classes reduces coupling and thus leads to a better design.
One of the main goals of a good class design is that of *localizing change*: making changes to one class should have minimal effects on other classes.

Cohesion

Cohesion describes how well a unit of code maps to a logical task or entity. In a highly cohesive system, each unit of code (method, class, or module) is responsible for a well-defined task or entity. Good class design exhibits a high degree of cohesion.
A cohesive method is responsible for one, and only one, well-defined task.
A cohesive class represents one well-defined entity.
High cohesion benefits a design by improving its readability and giving higher potential for reuse.

Efficiency

Java programs may be optimized to execute faster or to use less memory storage or other resources. Sorting algorithms provide some good examples of the ways in which the choice of algorithms and data structures affects performance. The topic of efficiency will be covered in more detail in Data Structures and Algorithms.