# Object Oriented Programming

Wednesday, January 13, 2010
8:51 PM

**States and Behaviors**

- The whole world can be divided into a series of interactions between one object and another
- All objects can be described in two ways -- through states and behaviors.

  **State:** What is the object? What does it look like? (adjectives).
  **Behavior:** What is its purpose? What can it do? (verbs).

**Redefinition of a Class**

- The blueprints for the construction of an object
- Defines what the object will look like and what it will do when it is constructed

**A "main-less" Class**

- Objects do not have main methods in them, because they are simply objects. A main method is like the "start" button to the whole system. An object is not a system -- it is a piece of the system.

  Example: A program is written to have a person get out of bed and make a cup of coffee. The person, bed, and coffee maker are all objects. The main method would be in another class (known as a **driver**) that basically commands the objects to interact in certain ways to produce the desired result.

  *Essentially, with the creation and manipulation of objects, the programmer gets to play God.*

**Standard Formatting of Objects**

Instance Variables

- Describe the state of the object
- Placed at the top of the code, outside of any method

Constructors

- Describes how to build the object
- When the code is run and it tries to construct this object, it needs to know how to do it.
- Constructors always match the class name, are always public, and have no return type.
- Basically initializes the instance variables -- either to default values or to values given as parameters

Behavioral Methods

- After the instance variables, constructors, and encapsulation are finished, the other methods of the object must be specified. We have figured out how to describe it and how to build it, and now we need to establish what the object can do.

**Creating the Driver**

- A driver program is used to manipulate the object
- Functions just like a normal main method
- The class with the main method does not usually function as an object. The other classes specify on the construction of the individual objects, but this class specifies how to build the system with all the objects together.

**Creating and Using Your Object**

Pointer Declaration

**Former Declaration Understanding:** ObjectType ObjectName = new ObjectType(necessary parameters);
**New Declaration Understanding:** ReferenceType PointerName = new ObjectType(necessary parameters);

- A **pointer** is a **reference** to a certain slot in memory
- Objects are only referenced -- they do not actually exist within the code
- Kind of like value on a bank note -- it doesn't individually hold any value but it can be cashed in for value

Passing the Reference

- Objects (such as arrays) can be modified within another method -- you aren't dealing with a copy of the value like with primitive data types.
- A reference to a memory spot is inputted as the parameter, but since the formal parameter becomes a pointer to the same reference, the actual object will be affected if it is modified in the method.

Trashing Objects

- An object that is no longer used (has no pointers) is picked up by the Java **Garbage Collector** and is deleted
- This avoids **memory leakage**, where massive amounts of memory are used up by the program unnecessarily
- Java takes care of all this, but if you programmed in C or many other languages you would have to watch out for this danger.

**<u>The . Operator and its Precedence</u>**

- In order to user an object's method, use the same syntax as you have been using with the Scanner object:
  pointerName.methodName(parameters)
- You can use the dot-operator to indicate a pointer chain that eventually will lead to where you want.

**<u>Organizing Code: UML Diagrams</u>**

- As you begin to work with objects and use them in your code, it can become confusing which objects use which other ones to function. It is therefore useful to map out the flow of your code and indicate the use of an object by a code with an arrow to that object. BlueJ compiler will do this by default, but you should become accustomed to doing it yourself.

**<u>Encapsulation</u>**

- Assume the programmers you will be working with are stupid, and that you are the only person who is ever going to understand your code
- You are going to need to protect your instance variables from foolish programmers who tamper with them (like setting a variable representing the age of the object to -50)
- The private keyword restricts their access to that part of the program, and instead they must use methods that you create, called **accessors** and **mutators**.
- A private variable can be accessed within that object's class, but it cannot be manipulated in another piece of code (like the driver) without accessors and mutators.

  <u>Accessor Method</u>
  - Query method that just returns a state of an object

  <u>Mutator Method</u>
  - Action method that modifies the state according to the inputted parameter

- While you are going to want the state itself to be private, you will want these methods to be public so that they can be accessed by the driver and by other external classes. When there is a method you want other classes to be able to use, be sure to define it as public. In general, if something doesn't need to be private, it should be public.

**<u>Static Data</u>**

- Distinguishes between **intrinsic** and **extrinsic** properties

  An **intrinsic** property, defined by an **instance variable**, is one that pertains specifically to that object. For instance, cars have all sorts of colors, so color would be an intrinsic property -- different for each object.

  An **extrinsic** property, defined by a **class variable**, is one that applies to the class as a whole. For instance, every penny has the same face on it. If the design of the face changes, it will change for every penny produced in the future. If a class variable is altered, every object with that class variable will undergo that alteration. Like instance variables, class variables can also be encapsulated.

- Since class variables apply to the class as a whole, they do not need an object to be created to be accessed. Thus, we can just use the className instead of the pointerName before the dot.

**<u>Top-Down Design</u>**

- Writing object methods out by header first to outline the code
- Once all the method headers and instance variables are written, the actual method bodies can be filled
- This helps to organize thought processes and code so that the programmer does not get bogged down in the longer and messier coding problems that often use objects

**<u>Reevaluation: *public static void main(String[] args)*</u>**

- The method is public -- all classes can access it
- The method is static -- it applies to all versions of the class (even though there is only one)
- The method is void -- it returns nothing.
- The method is named "main"
- The method takes a String array as a parameter
- The pointer to the String array is named "args"

**<u>Method Overloading</u>**

- You can create methods with the same name and different parameters

- Enables you to perform various actions with a calling -- sometimes organizes methods better
- Allows you to create multiple constructors based on the given input
- Cannot create same method name and have one be private and one be public unless you already have a parameter change
- Cannot create same method and have one static and one instance unless you already have a parameter change
- Cannot create same method name and have a different return type unless you already have a parameter change

**Good Examples**

```
public static void test()
public static void test(int x)
---------------------------------
public static void test()
private int test(int x)
```

**Bad Example**

```
public static void test()
private int test()
```

**Return of the Null Pointer Exception**
- Object manipulation often leads to many null pointer exceptions
- Remember that you cannot use an object that has not yet been created
- If you just state the beginning part of the declaration (ObjectType pointerName;) like you would for a primitive declaration, you will end up with a null pointer, as this pointer points nowhere.

**Shadow Variables**
- An instance variable and a local variable with the same name and type
- Referring to the variable by name will automatically refer to the local variable, which is "shadowing" the instance one

**"this" Keyword**
- When you want to refer to the object in its class you can specify the object by the word "this"
- This enables you to use the same of object variables in parameters and distinguish between the two by using the "this" keyword

```
public class example
{
        private int variable;

        public example (int variable)
        {
                this.variable = variable; //The object's variable should be set to value of the parameter "variable"
        }
}
```

**A whole new set of possibilities**
- Take another look at the whole API structure -- every one of those non-italicized hyperlink on the side scroll-down menu represents a class that you can use in your code.
- Scanner is just one of those many classes there for you to explore.
- Using API objects is just like using your own, only you can't change their implementations.

# Primitive Wrappers

Wednesday, January 20, 2010
4:59 PM

**Definition and Purpose**

- A wrapper class is a class designed to allow complex manipulation of data types
- Each wrapper class is identified by the full name of the data type they hold with a capital letter.

  Integer holds int
  Character holds char
  Boolean holds boolean
  Double holds double
- Wrapper classes basically can be thought of as candy wrappers -- the candy inside is the value and the wrapper holds all the information about it.

**Integer.parseInt(str)**

- An example of a wrapper method is the Integer class' parseInt() method, which pulls the next piece of the string as an integer.
- This is a static method since you don't need to define an integer to call this method
- Not all wrapper methods are static -- many apply only to the wrapped object

**Unwrapping a wrapper**

- To extract the primitive data type from the wrapper, simply use the wrapper's method primitive_data_typeValue().
- Examples:

  Integer Wrapper: intValue();
  Character Wrapper: charValue();
  Boolean Wrapper: booleanValue();
  Double Wrapper: doubleValue();
- Wrappers are fairly easy to work with, and they enable several features that would ordinarily be unavailable. We will delve into some of these features -- others you will have to explore yourself.

**compareTo()**

- Useful method to compare two wrappers
- Returns 0 if they hold the same value
- Returns a negative number if the calling wrapper is less than the parameter wrapper
- Returns a positive value if the calling wrapper is greater than the parameter wrapper

**equals()**

- Returns true if they are equal, returns false otherwise
- Useful only where you just want to test whether they are equal -- for comparison compareTo() should be used

**Designing a Wrapper**

- You can also design a wrapper class like the ones created in the API and create your own methods that suit your individual needs
- Creating a wrapper is just like creating any other object and encapsulating a single primitive value.
- Usually the wrapper classes available in the API will be sufficient, so this won't be necessary. But it's something to consider if ever you find a shortcoming of the wrapper classes.

# ArrayLists

Tuesday, February 16, 2010
12:04 PM

**Advantages of ArrayLists**

- Have no set size like arrays do -- enables adding to the array without any size restrictions
- Bubbles that form within the set from empty values are eliminated and the elements are automatically brought up to fill their slots.

**Disadvantages of ArrayLists**

- Arrays can hold any value, but ArrayLists can only hold objects
- It's ok though, because we can wrap primitive values and contain them in objects.
- ArrayLists are slower than Arrays, so if using an Array is just as convenient, don't use an ArrayList

**Declaration**

```
ArrayList<Type> name = new ArrayList<Type>();
```

**Assignment**

name.get(index); is used in place of name[index];

**Passing ArrayLists into and out of functions**

- Passed by reference like all other objects

**Warning when using ArrayLists**

- Whenever an object is deleted or added, contents of the list may shift around. Thus, you must be very careful when iterating through a list while adding to it -- you might accidentally perform an action on the same element twice or you might skip an element altogether.

**Arrays vs. ArrayLists Syntax Chart**

| | **Arrays** | **ArrayLists** |
|---|---|---|
| Accessing Size | arr.length | arr.size() |
| Accessing Data | arr[index] | arr.get(index) |
| Modifying Data | arr[index] = val | arr.set(index,val) |
| Adding to List | Not Applicable | arr.add(val) or arr.add(index,val) |
| Removing from List | Not Applicable | arr.remove(index) |

**For-Each Loop**

- Also known as the "enhanced for loop"
- Iterates through a collection (arrays, arraylists) 1 term at a time.
- Syntax:

```
for (collectionType variable : collection)
{
	//Do something to this variable
}
```

- This loop **cannot** be used to initialize or modify values -- only to access them to achieve another goal (like adding them together).

**Varargs**

- One of the more esoteric features of Java coding is variable arguments (varargs)
- Enables an unknown number of parameters in a method

- Syntax:
  ```
  public void foo (int x, String… s)
  {
          //Do something in this method
  }
  ```
- s is now a collection of strings that were placed after the int parameter -- basically functions as an ArrayList -- useful code if you want to use an enhanced for loop on s:
  ```
          for (String str : s) {}
  ```
- After the ellipses you cannot have any other parameters
- Methods using variable arguments should typically not be overloaded because this can cause confusion.

# Sorting and Searching

Monday, March 22, 2010
9:24 PM

**Sequential Search**

- Scans each element in the array sequentially until the element is found or the end of the array has been reached.
- The array does not need to be sorted for this to algorithm work.

**Binary Search**

- Parses the array and finds the side the element will be found in, then parses that array and continues until the element has been isolated.
- The array needs to be sorted for this algorithm to work.

**Bubble Sort**

- Slow sort that compares each element to its partner and swaps them if necessary
- Repeats this process until the entire list is sorted (no swaps are needed)

Comparisons Performed (teal = swapped elements, green = correct elements)

4 12 3 17 1
4 3 12 17 1
4 3 12 17 1
4 3 12 1 17

3 4 12 1 17
3 4 12 1 17
3 4 1 12 17
3 4 1 12 17

3 4 1 12 17
3 1 4 12 17
3 1 4 12 17
3 1 4 12 17

1 3 4 12 17
1 3 4 12 17
1 3 4 12 17
1 3 4 12 17

1 3 4 12 17
1 3 4 12 17
1 3 4 12 17
1 3 4 12 17

**Selection Sort**

- Searches for the minimum/maximum (depending on whether you want increasing or decreasing order) element in the array and swaps it with the position index (starting at the end of the array).
- The position index moves up and the next value is searched for to swap.
- Repeats until the position index has moved to the opposite end

Comparisons Performed (teal = position index, green = minimum value)

4 12 3 17 1
1 12 3 17 4
1 3 12 17 4
1 3 4 12 14
1 3 4 12 14

**Insertion Sort**

- Designates one end of the array as the "sorted section"

- Takes the element at the end opposite the sorted section and appropriately inserts it in the sorted array.
- Repeats until the sorted section covers the entire span of the array.

Comparisons Performed (teal = position index, green = sorted section)

```
4 12 3 17 1
12 3 17 1 4
3 17 1 4 12
17 1 3 4 12
1 3 4 12 17
```

**Merge Sort**

- Parses the array in 2 and recursively calls merge sort again on the two arrays
- Arrays are merged back together in a sorted format and a single sorted array is returned
- This splitting and rejoining of the array ends with a single sorted array

Merge Sort (yellow = split, teal = swap, green = merge)

```
[4 12 3 17 1]
[4 12 3] [17 1]
[4 12] [3] [17 1] //Array is completely split

[4 12] [3] [1 17] //Pieces are swapped if necessary
[3 4 12] [1 17]
[1 3 4 12 17] //Array is completely rejoined
```

**Big-O Notation**

- Systematic approach to approximate the speed of an algorithm for a very large array of numbers by estimating the number of steps required

| Notation | Definition | Meaning |
|---|---|---|
| O(1) | Constant | Regardless of the size of the array, the speed will always be the same and the same number of steps will be involved. |
| O(n) | Linear | The speed is directly proportional to the size of the array. |
| O($n^2$) | Quadratic | The speed is directly proportional to the size of the array squared. |
| O($n^3$) | Cubic | The speed is directly proportional to the size of the array cubed. |
| O(log n) | Logarithmic | The speed can be cut to less than an average linear process. |
| O(n log n) | Linearithmic | The speed can be cut to less than an average quadratic process. |

**Order of speeds from least to greatest: O(1), O(log n), O(n), O(n log n), O($n^2$), O($n^3$)**

| Sorting Algorithm | Speed |
|---|---|
| Sequential Search | O(n) |
| Binary Search | O(log n) |
| Bubble Sort | O($n^2$) |
| Selection Sort | O($n^2$) |
| Insertion Sort | O($n^2$) |
| Merge Sort | O(n log n) |

# Inheritance

Friday, February 26, 2010
5:40 PM

**Inheritance Concept**

- Say you wanted to create 2 classes -- a person class and a teacher class. How would these 2 be related?

  **Instinct:** Make a person object a private instance variable of teacher. Then encapsulate it and treat it just like any other instance variable.

  **Problem:** That means that every teacher "has" a person, just like they have a name, a subject they teach, etc, which are also probably instance variables. But a teacher doesn't HAVE a person; a teacher IS a person.

**Testing Relationships**

- Whenever 2 objects interact, their reactions can be broken down into the **is-a** and **has-a** relationships stated above.
- In general, if there isn't an is-a relationship between 2 interacting objects, then the relationship is a has-a one.

**Extends Keyword**

- To indicate an is-a relationship between 2 objects, use the extends keyword:

```
public class Teacher extends Person
{
        //Contents of Teacher class
}
```

- Because a teacher is a person, a teacher has all the same properties as any other person and can do all the same things. In other words, the Teacher class **inherits** all the characteristics of the Person class. This relationship is called **inheritance**.

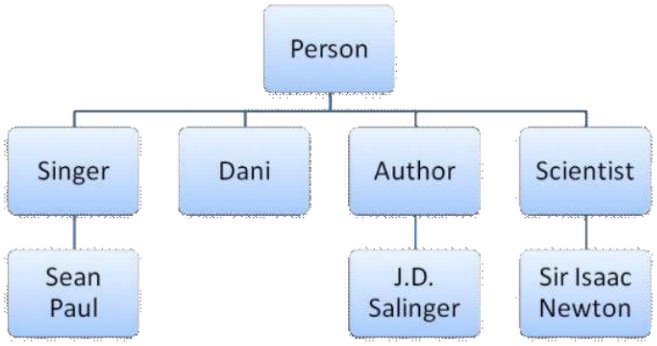
- A class that extends another class is a **subclass** of the first class. The first class is the **superclass** of the subclass.

- Classes can only extend **one** other class (i.e. Square cannot extend both the Rhombus and the Rectangle classes).

**Sample Hierarchy**

Take a look at the hierarchy on the left.
These are the facts that can be determined:

1. Dani is a person.
2. Sean Paul is a singer.
3. A singer is a person.
4. J.D. Salinger is an author.
5. An author is a person.
6. Sir Isaac Newton is a scientist.
7. A scientist is a person.
8. Sean Paul is a person.
9. J. D. Salinger is a person.
10. Sir Isaac Newton is a person.



**Polymorphism**

- Although the Teacher class inherits the Person class' states and behaviors, it can also override them to make them more suitable for use. Suppose, for instance, that the Person class has a printInfo() method that prints the person's name and age. The Teacher class might want to print the name, age, and subjects taught instead, so it defines its own printInfo() method. That method would be **overriding** the previous printInfo(), so when printInfo() is called on a Teacher object it uses the Teacher version, not the Person one.

- Constructors should be overridden -- if it is not overridden it uses the superclass' constructor

- Be careful not to confuse **method overriding** with **method overloading**!

**Super Keyword**

- The Teacher class' printInfo() method still might want to use the superclass' version of printInfo() in addition to its own modifications. To call the superclass' printInfo() method, simply use the syntax super.printInfo();

- To call the superclass' constructor, use the syntax super(parameters); in the **first line** of the subclass' constructor. If the superclass has a constructor **other than the default**, the subclass constructor **MUST** call super(parameters) as its first line. If the superclass **does have a default constructor**, the subclass constructor **MAY** call super(parameters), but it isn't required. If it does call it, however, it must be the first line.

- Basically the super keyword enables you to bypass the override and access the former method before it was replaced.

**<u>The King Class Of All Objects</u>**
- Try to trace the teacher class back up the hierarchy. A Teacher is a Person. A Person is a Mammal. A Mammal is an Animal. An Animal is an Organism. An Organism is a LivingThing. A LivingThing is an **Object**.

- If we are working with objects and breaking down the world into relationships between objects, than an Object is at the throne of the hierarchy. EVERYTHING is-a Object.

- Basically any class that does not explicitly extend another class is implicitly understood to extend Object. Thus, it inherits all behaviors and properties of the Object class.

**<u>toString() and equals() Overriding</u>**
- The Object class in the Java API has 2 very useful methods: toString() and equals():

- The toString() method returns a String representation of the object -- this is what will be printed if you simply call System.out.print(x) where x is an object. By default this will print the memory address of the object. However, you can override the toString() method so that it will print whatever you have your new method return.

- The equals() method compares 2 object pointers and checks to see whether they point to the same object. The equals() method in the String class overrides this and compares two strings to see if their contents are equal. You can override this method for your own comparison method.

  **Note:** *If you create a method in the Teacher class that accepts and Object as a parameter, you are overriding the method in the Object class. However, if you create a method that accepts a Teacher as a parameter, you are actually only overloading it.*

**<u>Private versus Protected Access</u>**
- The private keyword goes to extreme measures to cut off access to the data it protects to the point that literally only that class can access it. Even a subclass, which has all the properties of its superclass, still cannot directly access the variable
- The midpoint between private and public access is protected access. A protected variable can be directly accessed by that class and by any subclass, but can only be accessed through accessor and mutator methods by an external class.

**<u>Class Casting</u>**
- Because of the nature of is-a relationships, we should be able to have superclass pointers refer to subclasses as well. For instance:
  Person p = new Teacher(); should be a valid declaration, because p can be any person, and all we know is that p happens to be a specific kind of person -- a teacher.
- This idea is known as "**up-casting**." Up-casting does not require any special syntax -- you simply use the more general pointer to point to a specific instance of that class.
- However, what if you want to now use a method from the subclass with a superclass' pointer? You would need to send the object back down to the subclass level. This idea is known as "**down-casting**." Unlike up-casting, down-casting requires special syntax and can easily generate errors.
  <u>Syntax:</u>
  ```
  Superclass top = new Subclass(); //upcasting
  ((Subclass)top).subclassMethod(); //downcasting to access subclass' method
  ```

  <u>Example:</u>
  ```
  Person p = new Teacher(); //upcasting
  ((Teacher)p).addStudent(new Student()); //downcasts pointer p to a teacher pointer
  ```

and calls the teacher method addStudent

ClassCastExceptions:

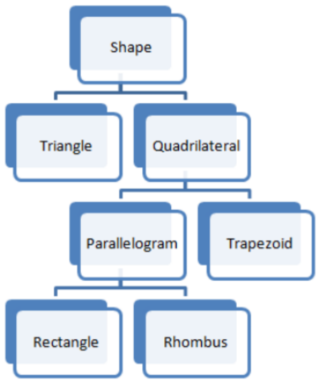
The type of error a down-cast will cause is a "class cast exception." This happens when an object is casted to a type that it doesn't fit, i.e. into a class that is that is not its own or its ancestor.

Example:

Assume that all pointers to all of these objects have shape pointers through upcasting.

The following declarations are valid:

```
(Shape)shape;
(Triangle)triangle;
(Quadrilateral)parallelogram;
(Trapezoid)trapezoid;
(Parallelogram)parallelogram
(Parallelogram)rectangle;
(Rhombus)rhombus;
```



The following declarations are invalid:

```
(Triangle)shape;
(Prallelogram)quadrilateral;
(Rectangle)prallelogram;
(Parallelogram)trapezoid;
(Rhombus)rectangle;
(Trapezoid)rhombus;
(Rectangle)shape;
```

**Instanceof Keyword**

- In order to prevent class cast exceptions, the instanceof keyword can be used to test whether a downcast will be valid.

Syntax:

```
if (X instanceof Y)
    (Y)X;  //will be valid.
```

Example:

```
Shape t = new Triangle();
Shape r = new Rectangle();

if (t instanceof Triangle)
    (Triangle)t; //will not cause class cast exception.

if (r instanceof Triangle)
    (Triangle)r; //r is not an instance of the Triangle class,
                 //so this line will never be reached, and thus no error will occur.
```

# Final Classes, Abstract Classes, and Interfaces

Wednesday, March 03, 2010
8:30 AM

**Final Classes**

- **Usage:** For a classification that is so specific that an object needs no further specifications of its properties from subclasses.
- Methods can be declared final that should **never** be overridden by a subclass.
- Classes can be declared final that should **never** have a subclass.

**Abstract Classes**

- **Usage:** For a classification that is so vague that an object can't be created based on that information alone.
- Methods can be declared abstract that should **always** be overridden by a subclass.
- Classes can be declared abstract that should **always** have a subclass.
- Only abstract classes can contain abstract methods.
- Abstract methods should not be given a designated body.
- If a subclass does not override the abstract method of a superclass, it too must be abstract.

**Interfaces**

- We have already discussed the "has-a" and "is-a" relationships. Interfaces introduce a new one: the "can-do" relationship.
- Interfaces are entirely abstract classes with no implementation whatsoever. They are solely used for organizational purposes to list the possible methods the objects can use.
- Just as an inheritance relationship is denoted by "subclass extends superclass," so too an interface relationship is denoted by "class implements interface."
- Unlike inheritance which limits each subclass to a single ancestor, an unlimited number of interfaces can be implemented by a given class.
- Subclasses automatically implement any interfaces that their superclasses implement.

  Comparable Interface
  - Commonly used interface with the compareTo() method
  - Enables methods to be defined with objects of type "Comparable" as the parameter for the compareTo() method

  *Note: Several interfaces will be introduced as we delve into more advanced Java concepts.*

**IF YOU ARE STUDYING FOR THE *AP COMPUTER SCIENCE A* EXAM,**
**SKIP TO THE GridWorld CHAPTER**

# GridWorld

Monday, March 08, 2010
11:40 AM

**Purpose of GridWorld**

- Created by the College Board to test the ability of coders to understand elaborate codes written by other people.
- Testable on the AP Computer Science A exam, but doesn't have much relevance to practical Java
- Fun tool for a lot of amusement if you know how to use it effectively.

**Downloading and Installing GridWorld**

1. Go to http://www.collegeboard.com/student/testing/ap/compsci_a/case.html
2. Download the zipped code file into the directory you use to store your programs.
3. Open BlueJ and click Tools --> Preferences
4. Open the Libraries tab and click Add
5. Select the gridworld.jar file in the GridWorldCode folder
6. Exit BlueJ to allow the preferences to be reset and then open it again to start using this library.

**Importing GridWorld Data**

Inside the framework folder you should find a package directory called info. Inside this should be another package directory called gridworld. Thus, the format for the import statements will be:

```
import info.gridworld.PackageName.ClassName;
```

**Explanation of the Grid**

An elaborate explanation of the API was produced by the college board here:

http://www.collegeboard.com/prod_downloads/student/testing/ap/compsci_a/ap07_gridworld_studmanual_appends_v3.pdf

# AP Computer Science A

Wednesday, March 10, 2010
7:57 AM

**Materials for Test Day**

You are given this quick reference guide:

http://www.collegeboard.com/prod_downloads/student/testing/ap/compsci_a/compsci_a_exam_appendix.pdf

**Test Information**

A complete list of what Java you need to know is here:

http://www.collegeboard.com/student/testing/ap/compsci_a/java.html

*I think all of it is covered in either Java Course I or Java Course II*

You have 1 hour and 15 minutes for the 40 question multiple-choice section. Along with topics from the course outline, the multiple-choice section includes at least five questions based on the AP GridWorld Case Study.

The free-response section contains 4 questions to be completed in 1 hour and 45 minutes. This section requires you to demonstrate the ability to solve problems involving more extended reasoning. Along with topics from the course outline, the free-response section includes one question based on the AP GridWorld Case Study.

The free-response questions from every year are published by the College Board and can be found here:

http://www.collegeboard.com/student/testing/ap/compsci_a/samp.html?compscia

Important Years:

2004 -- Test switched from C++ to Java

2008 -- Case Study switched from Marine Biology to Grid World