I. Review of Object Oriented Programming

A. Brief Review & Example

Concept/Program — Realization

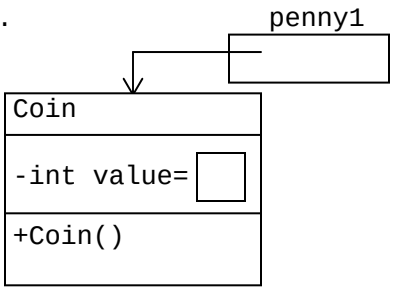
Class - description — Object - actual "thing" created "outside" the class

```
public class Coin
{
      public int value;

      public Coin()
      {
            ...
      }
...
}
```

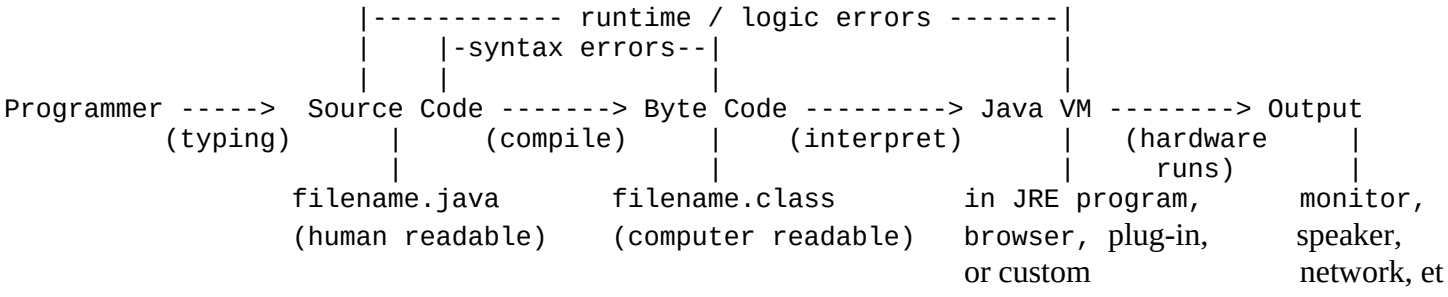
```
Coin penny1;         // declaration
penny = new Coin(); // instantiation
```

Diagram (Universal Modeling Language-UML)

```
object name (variable name) ....
object reference (unnamed box)

Class name (Type)

      fields

      constructor(s)
      & methods
```



B. Sequence of a Program & Introduction to IDE's



C. Full example

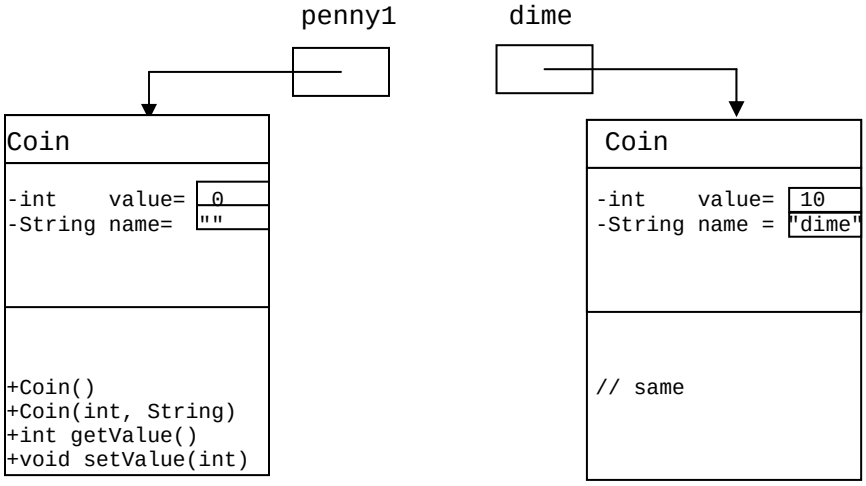
1. Class Definition

```
public class Coin
{
	// private class fields (variables)
	private int    value;
	private String name;

	// default constructor
	public Coin()
	{
		value = 0;
		name  = "";
	}

	// constructor with two parameters
	public Coin(int v, String n)
	{
		value = v;
		name  = n;
	}

	// accessor method (to access one field)
	public int getValue()
	{
		return value;
	}

	// mutator method (to change one field)
	public void setValue(int v)
	{
		value = v;
	}

} // Coin
```

2. Class Usage

```
public class UseACoin
{
	// class field variable
	private Coin penny1;                          // declaration
	private Coin dime1 = new Coin(10,"Dime");    // declaration and instantiation

	// application (no need a default constructor)
	public static void main(String a[])
	{
		penny1 = new Coin();   // constructor call to instantiate a new coin
		penny1.setValue(1);    // method call with 1 argument
		System.out.println(dime1.getValue());        // call to inherited (automatically) System class
	}
} // UseACoin

Notes:
    a) Syntactically correct but poor choice of class name UsaACoin --- should be UseCoin or TestCoin
    b) System.out.println(dime1.getValue()); will not work as expected -- must convert int to String by
       System.out.println("" + dime1.getValue()); --- details make all the difference in the world at this level!!
```

3. UML



Notes

a) The "-" means private, the "+" means public

4. Commenting

a) Three Java Commenting Options

(1) single line commenting

- Syntax: `// ...anything you need to say`
- Result: Anything after // is ignored by compiler to end of line
- Usage:
  - if used after statement, then explains usage/purpose of that statement
  - if used on its own line, then explains next few statements or code segment
  - if used on its own line, then put blank line *above* comment line

(2) multi-line commenting

- Syntax:
```
/* ... anything you need to say on
                    multiple lines...
*/
```
- Result: Everything between /* and */ is ignored by compiler
- Usage:
  - To explain complex segment of code that takes up more than one line
  - One its own lines, directly above code it explains
  - One or two blank lines above opening /*

(3) documentation commenting (JavaDoc utility)

- Syntax:
```
/** ... anything you need to say for use
          as "standard" code documentation...
*/
```
- Result: Everything between /** and */ is ignored by compiler
- Usage:
  - To explain feature of code for use in standard documentation (JavaDoc)
    - Classes
    - Methods (except main method)
    - Fields
  - One its own lines, directly above code it explains
  - One or two blank lines above opening /**
  - May contain special tags that start with the @ character...explained later
  - javadoc utility creates standard HTML documentation from these comments

5. Review Usage of TextPad and Introduction to SunOne Studio (Community Edition)

a) Show program, compile, run, & javaDoc using TextPad

b) Show program, compile, run, & javaDoc using SunOne Studio

c) Expectations for Programs to Be Turned In

- Hardcopy (printout) of each class unless specified otherwise. Each class must start on its one page unless multiple classes (complete files) can fit on a single page. New Currier font, 10 pt, no italics, no bold
- Appropriate and consistent commenting and indenting
- Be prepared to demonstrate to Mr. M. your program (submitted version) on the teacher's station

D. Common Errors & Mistakes (stupid programmer tricks)

1. Compiler / Syntax Errors

- breaking the "grammar" rules, easy to fix as compiler tells you
- common examples:
    i. forgetting a semicolon (this is not the same as adding extra semicolons!!)
    ii. misspelled words
    iii. incorrect case (capitalization)

2. Runtime Errors

- unhandled exception
- usually due to not checking all possible values -- defensive programming
- results in a crash of program
- common examples
    i. nullPointer exception -- object declared but never initialized or instantiated
    ii. divide by zero
    iii. forgetting to initialize a variable either in constructor or at declaration

3. Logic Errors

- you didn't think carefully enough -- you assumed you KNEW the answer
- may cause crash (hopefully) or just plain wrong answer/result
- most dangerous and difficult to track down
- usually can't prove correct unless using exhaustive testing or mathematical proof
- common examples
    i. off-by-one error -- especially using arrays since start at subscript 0
    ii. incorrect formula -- Area = 2.0 * Math.PI * radius
    iii. incorrect variable usage (worse yet, Area = 2 * Math.PI * radius)
    iv. roundoff errors - fundamental limits of data storage

4. Avoiding Errors

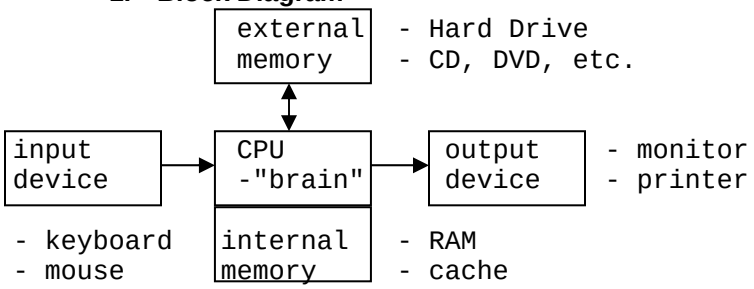
- Syntax Errors
    - Know Java syntax!
    - Use style conventions (indenting, commenting, variable names, etc)!!
- Runtime Errors
    - Know OOP concepts!!!
    - Passing argument variables to parameter variables in method calls!!!
    - Habits!!! (always initialize all variables, know Class types, know instantiations)
- Logic Errors

- Plan FIRST!!!!... 10 years of education and fast food has taught you to "get working right away and get it over with"...this will be your downfall. Be prepared for 6 weeks of boredom and lecture notes.
- Know defensive programming!!!!
    - assume you are WRONG
    - Use easy, known test data to check your BASIC understanding of problem
    - Test "boundary" conditions...be paranoid!!!!
- Know debugging techniques!!!!
    - Pinpoint -- determine which methods your program is running and in what order (don't just assume)
    - Snapshot -- determine variable/object values at various times
    - Use either System.out.println() or IDE debugger

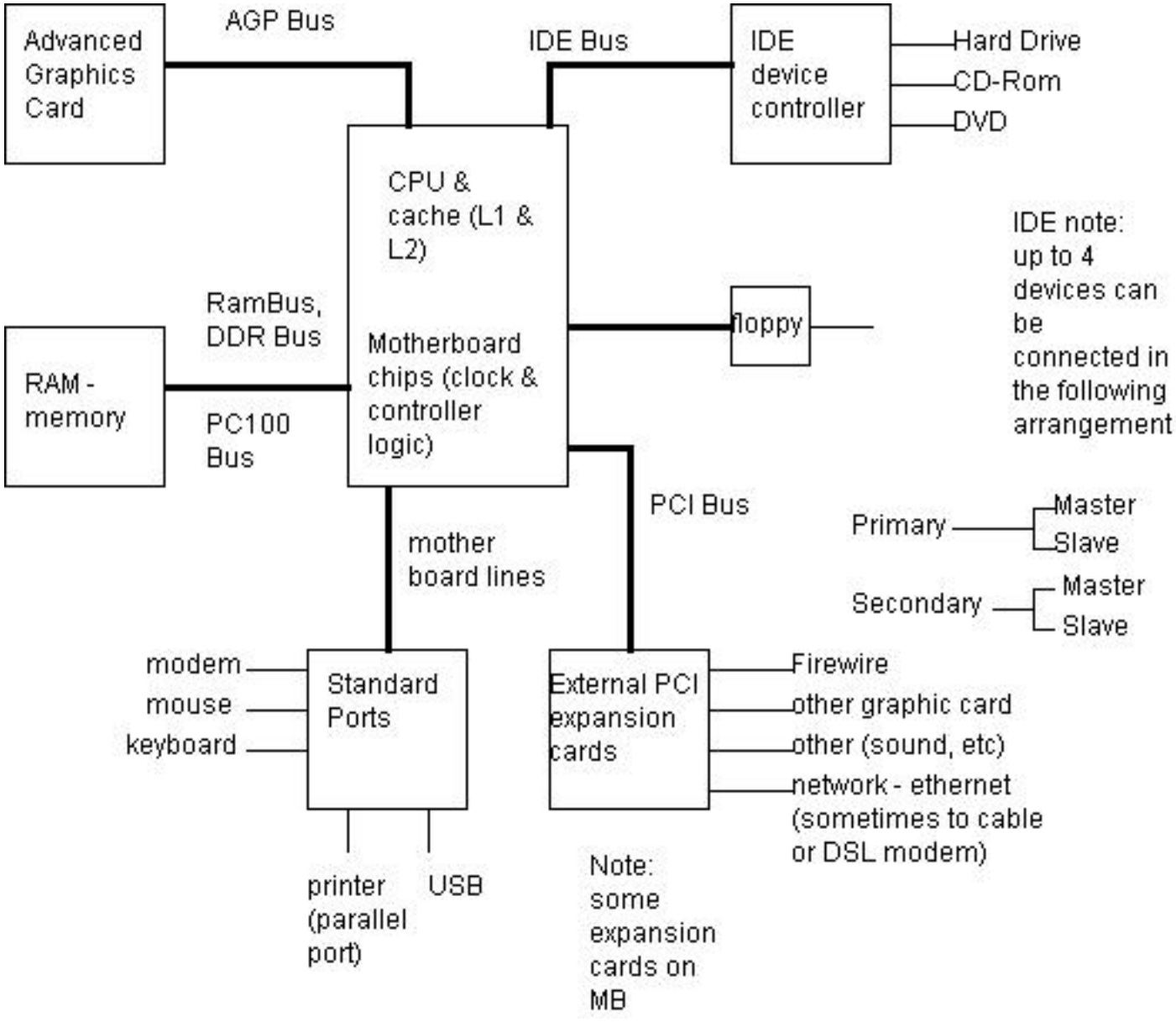
II. Digital Computer Hardware and Basic Concepts

A. Diagrams

1. Block Diagram



2. Schematic Diagram



B. Abstraction versus Realization

Data Abstraction Terms (coding/meaning)	Realization (measurable)
1. bit ("0" or "1", "off" or "on")	switches (CD's: no pit = 0, pit =1) --> used in memory, storage
2. 1 byte = 8 bits = 256 "states" or values	
3. 1 kilobyte (KB) = 1024 bytes (sometimes shortened to 1000 bytes)	
4. 1 megabyte (MB) = 1024 KB (sometimes shortened to 1000 KB)	
5. 1 gigabyte (GB) = 1024 MB ...	
6. 1 terabyte (TB) = 1024 GB = 8796093022240 bits	

Action Abstraction Concepts/Tems	Realization (measurable)
1. 1 hertz (Hz) Clock speed	1 "flip" per second ... flip may be 1 bit or numerous bits together
2. 1 kiloHertz (KHz) = 1000 Hz	Note: our vision is "updated" at 30 Hz... anything less will
3. 1 megaHertz (1MHz) = 1,000,000 Hz	"flicker". However our eyes are sensitive to GHz... we
4. 1 gigaHertz (1GHz) = 1 billion flips per second	perceive differences in this range as shades of color.

Basically, computers now use "multiple" clocks that are synchronized to one "master" clock. This allows for optimization and efficiency depending on requirements of hardware. Example is "sleep mode" for laptops where the CPU may slow its clock(s) to save energy/battery.

III. Basic Terminology & Concepts in OOP

A. Divide & Conquer

1. **Break overall problem down into small, manageable tasks**

2. **Allows for multiple programmers to solve one problem**

B. Encapsulation -- each object knows about itself and is separate from other objects

1. **Each object "knows"**

   a) **what it IS -- fields**

   b) **what it can DO -- methods**

   c) **how to interact --**

      (1) data hiding = private

      (2) interface = public

C. Generality -- also known as "can be applied to a lot of situations or problems"

1. **A *Class* is a *Type* of problem -- it solves a general problem (i.e. the quadratic function solves a type of problem (equations of the form ax^2 + bx + c = 0)**

2. **An *object* is a specific *instance* of a problem -- 2x^2 -3x +6 = 0 is an instance of a quadratic equation**

D. Extensibility - the ability to add or include features easily

1. **Two kinds of relationships:**

   a) **inheritance**

      (1) also know as a "is a" relation. A PokerDealer "is a" PokerPlayer. Therefore class PokerDealer extends PokerPlayer

      (2) allows for a "hierarchy" to organize classes into useful categories. ex. animal classification system, JFC

      (3) extended classes

         (a) inherit all the abilities (fields & methods) of the "super" or ancestor classes

         (b) may overload ancestor methods (polymorphism)

         (c) may write additional methods and/or fields as needed

b) association -

(1) also known as a "has a" relation. A PokerGame "has a" Deck. Therefore, we can create or instantiate as many objects from a class to use as needed.

(2) used for "code reuse"--why reinvent a program when we can just "use it?"

2. a *Class* can be *extended* or specialized to answer a more detailed question/problem -- `class PokerApplet extends Applet` **can do everything an Applet can (i.e. internet access) but does something more specific in that it can play poker over the internet**

3. a *Class* can *implement* some other programmer's "agreed-upon" class where the other programmer describes what needs to be written but not <u>how</u> to do it.
-- `class Poker ... implements ActionListener` **forces the Poker programmer to write a method called** `actionPerformed(ActionEvent e) {}` **but doesn't say how or what to write inside it.**

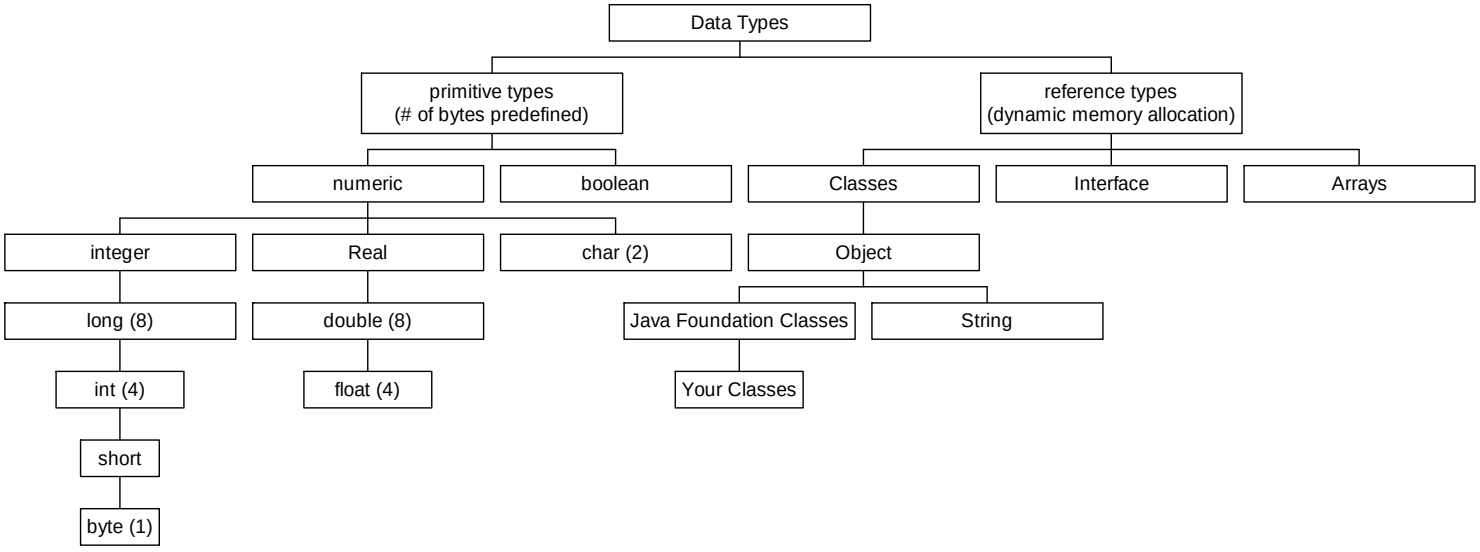
E. Abstraction -- The Overarching Principle of Above Concepts

1. "What makes a human different than a computer"

2. Discovering the essential features (nouns & verbs) of a class (fields & methods) in order to solve a question/problem. This may entail several "layers" of abstraction (i.e. multiple related classes) and is best learned through experience (i.e. failure many times)

IV. Java Syntax

A. Data Types



B. Variables

1. 2 Types

a) field variables belong to the class

b) local variables belong to a scope within a method

2. Syntax

```
[final] Type variableName [= initializer];
```

where

a) `[...]` **is optional**

b) `Type` **is one of the Data Types above**

c) `variableName` **is chosen by the Programmer and must:**

(1) consist only of digits, letters, and underscores

(2) must begin with a letter or underscore

(3) case sensitive

(4) has a scope -- it exists from point of declaration to nearest end of block i.e { ... }

(a) a `final` variable must be initialized only once at point of declaration (which makes it a definition)

(5) will be graded according to style:

(a) all letters lowercase

(b) except:

(i) `final` variables are all capitalized (also known as "constants")

(ii) the first letter of words following first word in variableName must be capitalized

(c) meaning or purpose in program

3. examples:

- `int n;` `// declaration only`
- `double x = 0;` `// declaration & initialization (definition)`

- `String harry = "Harry Handsome";`
  `// object declaration & instantiation with assignment/initialization to String literal`

- `Rectangle box = new Rectangle(5,10,20,30);`
  `// object declaration & instantiation (definition) using a constructor with 4 integer arguments passed by value to 4 integer parameters`

- `int [] perfectSquares = {1,4,9,16,25};`
  `// array reference/object declared & instantiated (automatically/implicitly to size 5) and initialized with 5 integer values by "block initializer".`

- `final int MAXRANGE = 100;` `// constant`
- `final float _EPSILON = 0.0001;`
  `// constant (the starting underscore implies it is a "system" variable)`

4. local variables vs. class fields -- scope and qualifiers

a) Definition & Usage

(1) a local variable is for temporary use in a method and therefore cannot be accessed outside its scope

(2) fields are variables defined in the class (outside of all methods)

(3) the scope of a variable is the region (usually denoted by {...}) of a program in which the variable can be referred to by its variableName.

(4) to access a field outside of its scope (i.e. class), it must be qualified (using dot notation). Assuming the field is either `public` or `protected` (not `private`)

(5) within its scope (i.e. within its class) a field has an implicit qualifier, namely `this` (ex. `this.penny1`)

b) Notes

(1) It is a syntax error to have 2 identical local variables within the same scope or overlapping scopes

(2) variableNames should NOT be distinguished by case alone!

(3) duplicate fields & local variableNames:

(a) can overlap scopes

(b) cannot be both declared inside same scope -- why not?...ans: fields are class scope, locals are method scope

(c) this is called "shadowing"... which can be useful but very dangerous if not done carefully!! If you intend to do this, be prepared to explain why.

(4) it is a syntax error to have two duplicate field variableNames within one class scope.

c) Examples

```
public class Test
{
      private int x;                // field variable,
                                    // scope is the Test class,
                                    // has implicit Test class qualifier this.x

      public Test (int x)           // local variable (aka parameter -- has method scope)
      {
            int    x;               // illegal as same scope as parameter

            String y;               // local variable object, method scope

            int    y;               // illegal, already declared local variable y

            for (int i=1; i<x; i++) // ok, using local variable x, not field x
            {
                  y = "hello";      // ok, just using local variable y
            }
      }


      public Test()           // default constructor
      {
            int x = 5;        // local variable x defined, shadows field x (hides the field)
                              // different scope the above constructor

            x = x + 1;        // OK since just modifying local variable

            x = x++;          // OK but what will x be after this?
                              // i.e. don't mix unary & binary expressions

            this.x = x;       // OK, private field x is being set equal to local variable x
      }
} // class Test
```

C. Expressions

1. Dfn: An expression is one of the following:

a) a literal (string or character or numeric) -- i.e. "this is a string literal", 'z', 23.0

b) a variable (local or field--perhaps qualified)

c) a method call (i.e. that returns a value or reference to an object)

d) combination of subExpressions acted on by operators

2. Examples:

```
a) '5'            // a character literal

b) x              // variable (type unknown)

c) Math.sin(x)    // method call (specifically a static class method)

d) x + Math.sin(x)            // two subexpressions acted upon by the +
operator (Note: both subexpressions must be of compatible types)

e) x++                        // a variable acted upon by the ++ operator

f) x == y                     // 2 variables acted upon by the == operator
(returns a boolean value)

g) x == y && (z>0 || w<0)     // 4 variables, 5 operators

h) p.x                        // a variable, specifically a "qualified field
variable", q is the "name of the object" or "reference to the object" or
"name of the instance" and x is the class field

i) v[i]                       // variable, specifically an array element
where v is the array, i is the index

j) e.getSalary()              // method call of an instance/object named e.
```

3. Operators

a) Three types of operators

(1) unary -- acts upon only 1 variable

(a) prefix -- operation is done before all other operators

(i) ++x, --x, -x, +x (Note: +x is same as x, -x is the negation of x if it exists)

(b) postfix - operation is done after all other operators are finished

(i) x++, x--

(c) other

(i) . Access class feature ("qualifier operator")

(ii) [] Array subscript

(iii) () method call

(iv) ! boolean NOT

(v) ~ bitwise not

(vi) (TypeName) Cast to another (super) class

(vii) new Object allocation/instantiation

(2) binary -- works on two expressions

(a) * multiplication

(b) / division or integer division

(c) % integer remainder (modulus)

(d) + addition

(e) - subtraction

(f) << shift left

(g) >> arithmetic shift right

(h) >>> bitwise shift right

(i) < less than

(j) <= less than or equal to

(k) > greater than

(l) >= greater than or equal to

(m) == equal to

(n) != not equal to

(o) instance of tests whether an object's type is a given type or subtype thereof

(p) & bitwise and

(q) ^ bitwise exclusive or

(r) | bitwise or

(s) && bitwise "short circuit" and (also called a "logical and")

(t) || bitwise "short circuit" or (also called a "logical or")

(u) = assignment

(v) op= assignment with binary operator (op is one of +,-,*,/,%,&,|,^,<<,>>,>>>)

(3) ternary -- operates on 3expressions, Java has only 1 and is only rarely used

(a) testExpression ? expression1 : expression2

(b) same as:

```
if (testExpression == true)
      return expression1
else
     return expression2
```

b) Precedence -- normal math precedence with ( ) as precedence override

c) Notes

(1) most operators are left-associated

(a) ex. x - y + z is same as (x - y) + z which is the same as mathematics

(2) exceptions (which are right-associated) are

(a) unary prefix and others -- examples include +x, -y, ++x, --x, !x, ~x, (int)x, new X()

(b) assignment and op= (ex. x = y = Math.sin(j) is same as x = (y = Math.sin(j))

see page 123 in Big Java for exercises.

D. Class Declaration

1. Syntax of Class declarations

```
[public | private] [abstract | final ] class ClassName [extends SuperClassName ]  [implements InterfaceName1, InterfaceName2, ...]
{
      feature1;
      feature2;
...
}

where
      feature is either:
            1) a declaration of the form:
                  modifiers field | constructor | method | inner class
      or
            2) an initialization block of the form:
                  [static] { body }
```

2. Notes on Class declarations

a) the underlined items were introduced in Honors Java Programming course but not always defined or explained

b) the [...] is optional syntax

c) the | means "exclusive or" (i.e. either choose one or the other but not both options)

d) `abstract, final, extends, implements, inner class` **and** `static` **will be discussed later.**

e) modifiers are: (may choose one of `public, private,` **or** `protected` **along with either one or both of** `static` **and** `final`**)**

(1) public -- make this feature available to all other classes

(2) private -- make this feature only available to this class

(3) protected -- make this feature only available to this class, subclasses (descendents) and classes in same package (directory). Note: protected is not used in this course and is of limited value

(4) static -- defined for this class, not for each instance of the class. (i.e. every object of this class has the exact same--1 and only 1--feature)

(5) final -- (same as a constant) has the following three properties:

(a) a field that cannot be changed once it is initialized at the point of declaration

(b) a method that can't be overridden by another subclass (i.e. defeats polymorphism)

(c) a class that can't be extended (i.e. no subclasses)

3. field declarations

a) Syntax: `[modifiers] Type variableName [= initializer];`

b) 2 types of fields

(1) instance fields -- each object has its own copy of the field

(2) static fields -- only 1 per-class copy. Also known as "class fields"

(a) Note1: static fields are useful for "sharing" common information between all instances of a class

(b) Note2: This breaks "encapsulation." So, to avoid accidental changes, use as `static final.` Therefore it can be set only once by original programmer

(c) Note3: static variableNames use, by convention, all capital letters

(d) Note 4: there are three ways to initialize static fields:

(i) do nothing ... default values will be supplied by JVM... poor programming and will be major loss of points!!!

What are default values for int, long, float, double, char, boolean, byte, short???

(ii) explicitly at point of declaration/definition :
i.e. `public static final double PI = 3.141592;`

(iii) static initialization block: (not used in this course)

```
public class Test
{
      private static int X;   // field X is static

      public Test()           // constructor
      {
      ...
      }
      ...
      static { X = 0;}        // static initialization block for field X can be anywhere.
      ...
}
```

4. constructor declaration

a) Syntax

(1) syntax for <u>declaring</u> a constructor

```
[modifiers] ClassName ( [Type parameter1, Type parameter2, ...] )   [throws ExceptionType1, ExceptionType2, ...]
{
      body
}
```

(2) **syntax for <u>calling</u> a constructor** (or using the constructor from another class). AKA "invoking" or "instantiating"

```
new ClassName ( [argument1, argument2, ... ] );
```

b) Comments

(1) `throws` is used for "error handling" and will be discussed later

(2) when invoking a constructor, the arguments and parameters must match in:

(a) compatible Type (i.e. the same Type or can be "cast" to the same Type)

(b) same number of parameters as arguments

(c) same order of parameters and arguments

(3) when an object (instance) is constructed, the following actions take place:

(a) all fields are given default values

(i) `0` for numbers

(ii) `false` for booleans

(iii) `null` for reference/object fields)

(b) initializers & initializer blocks are executed in the order in which they appear

(c) body of constructor is executed/run

(4) when a class is loaded into the JVM the following actions take place:

(a) all static fields are initialized with default values

(b) initializers and initializer blocks are executed in the order in which they appear

(5) one constructor can call another constructor in the same class by the invocation:

(a) `this ( [ argument1, argument2, ... ] );`

(b) However, the invocation must be on the first line of the body of the calling constructor.

(c) Example:

```
private Coin(int v)                 // no one can use this constructor directly
{
       value = v;
}

public Coin (int v, String n)       // second constructor which can be used by others
{
       this(v);                     // calls the above constructor,  value now equals v (note: on first line)
       name = n;
}
```

(6) one constructor may call the constructor of the "parent" or "super" class (that the current class `extends`) by the invocation:

(a) `super ( [argument1, argument2, ... ] );`

(b) However, the invocation must be on the first line of the body of the calling constructor

c) Special references (i.e. Specialized Java Classes)

(1) arrays

(a) Syntax:

```
new ArrayType [ = {initializer1, initializer2, ... } ];
```

(b) example:

```
new int [ ] = {1,4,5, 16, 25};
```

(i) Note1: this is an "anonymous" array...no variable name is attached

(ii) Note2: the "int [ ]" is part of the ArrayType whereas the "[= { ...}]" is optional

(2) String class

(a) Construction

```
String name1 = "Rob";                 // one String object referring to a literal constant
                                      // string "Rob", this is an implied constructor invocation
                                      // unique in Java

String name2 = "R" + "o" + "b";       // one String object constructed from 3 anonymous objects

String name3 = new String("Rob");     // String reference to a new String object constructed to
                                      // contain a literal constant string "Rob",  this is a
                                      // same constructor call as for name1

String name4 = name3;                 // name4 is a new reference to the exact same object
                                      // as name3
```

(b) Notes

(i) All Strings are immutable -- once instantiated & initialized, a String object can never be changed. However it can be copied. In other words, after instantiation, there are only "accessor" methods but no "mutator" methods.

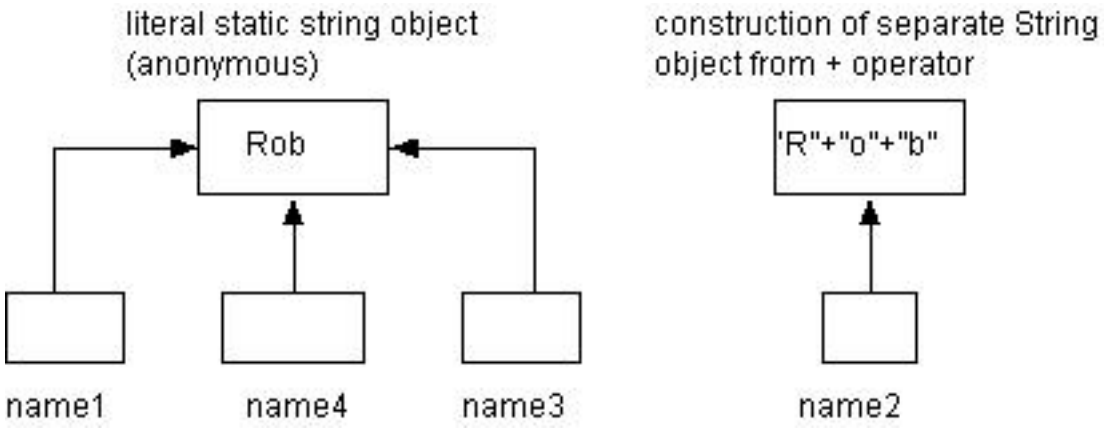
(ii) Do NOT compare Strings using ==. This leads to numerous problems as the "==" operator is meant for primitive Types.

(iii) Example:

(a) `if (name1 == name2)` *// returns false*

(b) `if (name3 == name4)` *// returns true*

(iv) Why?



(v) to compare String objects use the `equals()` method for each String object i.e.

```
if (name1.equals(name2))      // returns true since
                              // equals() method compares
                              // contents, not references
```

(vi) This warning holds true for comparison of any object. Ergo == operator compares if the references are the same, not contents.

(vii) In order to compare contents of the objects (i.e. field values the same), must use the `equals()` method.

(viii) However, comparison of contents is a complex topic and will be addressed only if we have enough time

(ix) Finally, the empty String `""` is not the same as the `null` String

```
String name1;
String name2="";
if (name1 == null)      // returns true
if (name2 == "")        // return true but dangerous! why?
if (name2 == null)      // returns false;
```

(a) the "" is a valid String object (i.e. a size 0 String)

(b) the `null` String means that a String variable has been declared but not instantiated or referenced properly (i.e. nullPointerException error)

5. method declaration

a) Syntax

```
modifiers ReturnType methodName ( [ [final] ParameterType parameter1,
                                    [final] ParameterType parameter2, ...]
                                  )
                                  [throws ExceptionType1,
                                          ExceptionType2, ... ]
{
      body
}
```

where

(1) ReturnType is any Java Type or void (i.e. no return value)

(2) modifiers are: (may choose one of **`public, private,`** or **`protected`** along with either one or more of **`static, final,`** and **`abstract`**)

(a) public -- make this method available to all other classes

(b) private -- make this method only available to this class

(c) protected -- make this method only available to this class, subclasses (descendents) and classes in same package (directory). Note: protected is not used in this course

(d) static -- defined for this class, not for each instance of the class. (i.e. every object of this class has the exact same--1 and only 1--method).

(i) also known as "class methods" as opposed to "instance methods" since the method belongs to the class as a whole

(ii) Example: Math.sqrt(x) // x must be either a static field or literal value

(iii) Note1: static methods can only affect static fields (not instance fields) and static parameters (cannot use the implicit instance qualifier "this")

(iv) Note2: the historical choice of term "static" is poor...as it implies that it cannot be changed, which is not true...it can change fields but they also must be "static"

(e) final -- has the following properties:

(i) the method cannot be overridden, (i.e. not overloaded or extended) by any subclass method (i.e. another programmer cannot "hide" your method behind his/her own method of the same signature)

(a) Dfn: a "method signature" consists of the methodName, parameterTypes, and parameter order but not the returnType, modifiers, or ExceptionTypes.

(ii) defeats polymorphism and therefore, in general, should be avoided

(iii) Example:

```
(a)   public final boolean generatePassword()
      {
            ...
      }
```

(b) the programmer did not want another programmer to be able to write a different generatePassword() method that might be used to generate "poor" passwords...prevent hacking

(f) abstract -- has the following properties

(i) forces another programmer to <u>extend</u> the method (and class)

(ii) has no method body (i.e. not even {...})

(iii) any class with any abstract methods cannot be instantiated (since an object won't know how to execute a non-existent method body). Therefore abstract methods (& classes) can only be extended

(iv) Example:

```
public abstract class BankAccount
{
   private String password = "";

   public abstract String createPassword() // another programmer must supply the body
}

public class SavingsAccount extends BankAccount
{
      // note: at this point we have access to password via qualifier "super.password"

      private String password = ""; // note this "overrides" BankAccount's password
                                    // but still can distinguish by using
                                    // this.password (instance field) or super.password

      public SavingsAccount()
      {
            super();                // invoke BankAccount's constructor (not shown)
      }

      public String createPassword()   // forced implementation of abstract BankAccount method
      {
            password = "secret";    // using this.password (not super.password)
            return password;
      }
}
```

(3) throws and ExceptionTypes will be discussed later

b) Notes

(1) Generic method declaration is not covered in this class...only Generic method call/invocationU...see later lecture.

(2) if ReturnType (other than void) is specified, then the `return` statement must:

(a) be "guaranteed" to be executable (otherwise a compiler warning)

(b) include an expression whose Type is "compatible" with ReturnType

(c) the method exits as soon as the return statement is encountered.

(3) do NOT put final and abstract in same class since:

(a) `abstract` implies the class must be extended

(b) `final` implies the class cannot be extended

(4) `[final] ParameterType parameter`

(a) ** Numerous common mistakes!!!

(b) ** Explanation: All arguments from calling (invoking) method are passed to the corresponding parameter by value (i.e. the value of the argument is copied into a new memory location for use as a parameter)

(i) for primitives, cannot change argument via the parameter (i.e. can't change "backward")

(ii) for references (objects), cannot change original argument reference but can change original argument reference (object) fields via mutator methods (i.e. setXXX()) and public fields & methods (i.e. normal class interfaces)

(c) Examples:

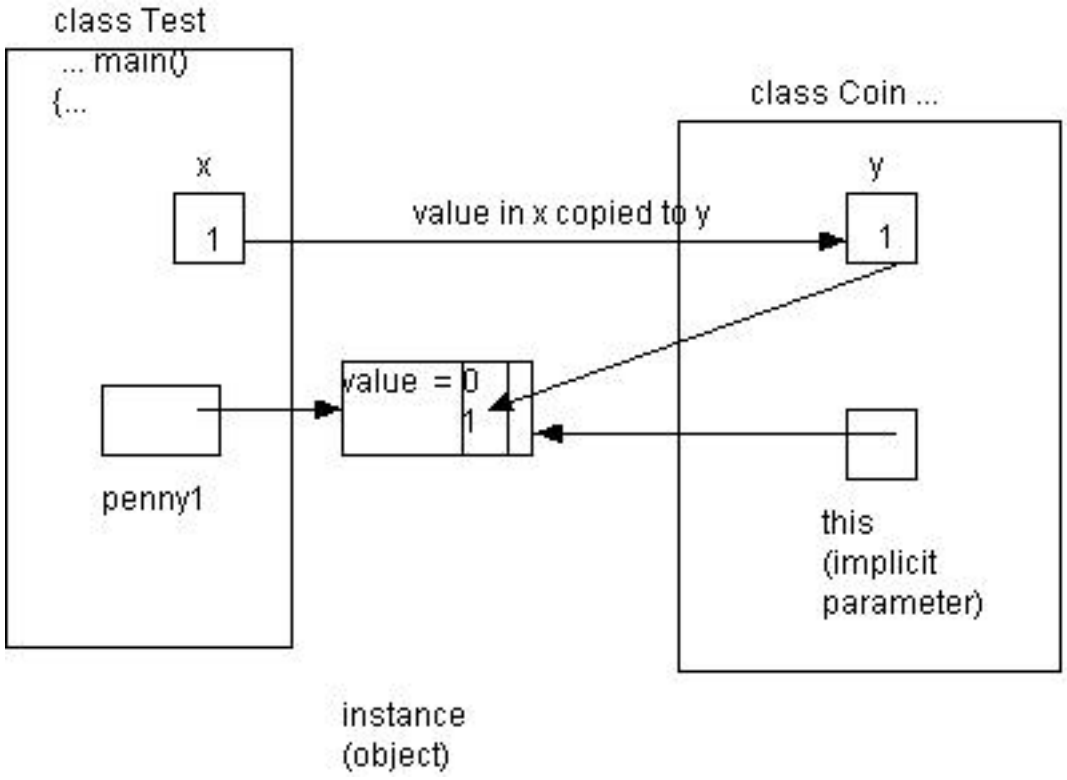
Example 1:

calling program/method

```
public class Test
{
      public static void main(String a[])
      {
            // local variables (not fields)
            static int  x      = 1;
            static Coin penny1 = new Coin();
            ...
            penny1.setValue(x);     // invoking method with argument x
      }
}
```

invoked program/method

```
public Class Coin
{
      private int value = 0;        // field variable
      ...
      public void setValue(int y)   // parameter y (local variable, method scope)
      {
            value = y;              // same as this.value = y;
      }
```



Example 2: (note: primitive variables not shown)

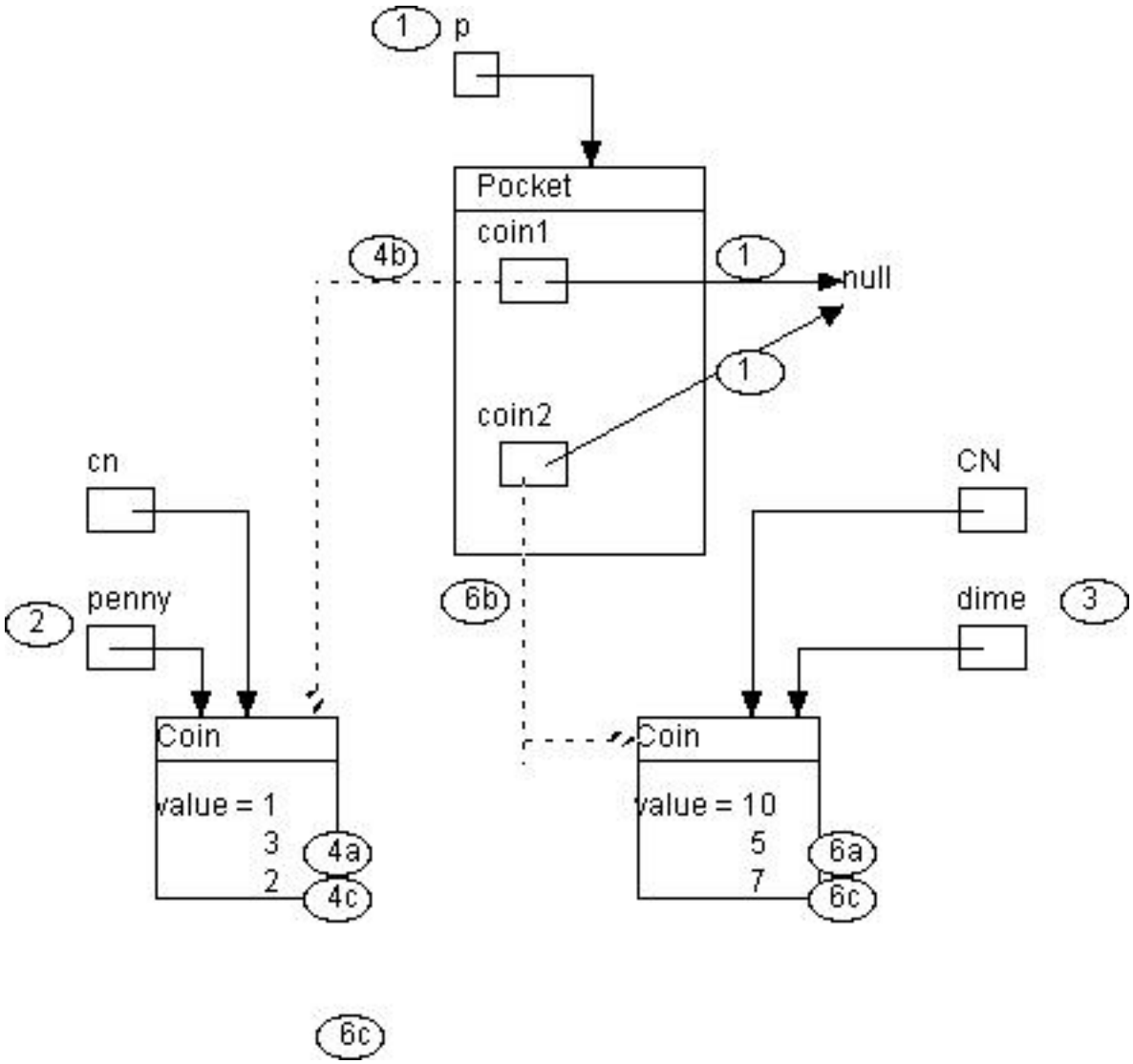
calling program/method

```
public class Test
{
      public static void main(String a[])
      {
            Pocket p;               // holds up to 2 Coins
1)          p = new Pocket();       // create a new pocket
            Coin penny,dime;        // two coins
2)          penny = new Coin(1);    // the penny object has value 1
3)          dime  = new Coin(10);   // the dime object has value 10
            ...
4)          p.setCoin1(penny);      // put a penny into the pocket's first coin
5)          penny = null;
6)          p.setCoin2(dime);       // put a dime into the pocket's 2nd coin
            ...
      }
}
```

called program/method

```
public class Pocket
{
      private Coin coin1 = null;    // first coin in pocket
      private Coin coin2 = null;
      ...
4)    public void setCoin1(Coin cn)
      {
4a)         cn.setValue(3);
4b)         coin1 = cn;             // same as this.coin1 = cn
4c)         coin1.setValue(2);
            coin1 = penny;          // syntax error (Pocket doesn't know about penny variable)
            coin1 = coin2;          // Ok, but dangerous as both slots refer to same coin
            coin1 = cn;             // Ok, corrected above problems
            cn    = null;           // OK since don't need parameter reference anymore
      }
      ...
6     public void setCoin2(final Coin CN)
      {
6a)         CN.setValue(5);         // OK but dangerous as just changed the value of dime
6b)         coin2 = CN;             // same as this.coin2 = dime
6c)         coin2.setValue(7);      // OK but dangerous as just changed the value of dime
            coin2 = coin1;          // OK but now the dime is a penny;
            coin2 = CN;             // back to dime and coin2 having value 5
            CN    = null;           // illegal as CN can't reference anything other than dime
      }
}

public class Coin
{
      private int value;
      ...
      public boolean setValue(int x)      // the boolean return value can be ignored by
      {                                   // calling program (i.e. Pocket & Test classes)
            value = x;                    // same as this.value = x
            return true;                  // required since returning a boolean
      }
}
```

Note1: penny & dime are argument references. cn and CN are corresponding parameter references.
Note2: penny1, cn, and coin1 references are copies (i.e. they point to same place or address). But the object itself is NOT copied. (Think of MS Windows shortcuts--1 file but multiple icons that can access the file)


1
p
Pocket
coin1
4b
1
null
1
coin2
cn
CN
2
penny
6b
dime
3
Coin
value = 1
3
2
4a
4c
Coin
value = 10
5
7
6a
6c
6c

6. Interface declaration & usage

a) Syntax for declaration

```
[public] interface IntefaceName [extends InterfaceName1, InterfaceName2, ..]
{
      feature1;
      feature2;
      …
}
```

where:

1. each feature has the form:
    a. modifiers method | field
        i. modifiers are public, static, and/or final
        ii. modifiers are never necessary because:
            1. methods are always public
            2. fields are always public static final
        iii. method declaration has the form:
            1. Type methodName (parameter1, parameter2,…);
        iv. Field declaration has the form:
            1. Type variableName = initializer;

b) Example

```
public interface Measurable
{
      double CM_PER_INCH = 2.54;          // assume public static final

      int getMeasure();                   // assume public
}
```

c) Notes

(1) Interfaces can be thought of as “Abstract classes on steroids”

(2) Interfaces are “agreed upon” set of method signatures that will be implemented by the individual programmers.

(3) Converting from Class to Interface is legal (and automatic) as long as the Class implements the Interface….”upcasting is automatic”

(4) Casting or converting from an Interface to a Class is legal (but not automatic) as long as the Class implements the Interface and the programmer KNOWS the object is of the desired class. “downcasting must be down explicitly and carefully”

(5) See API List Interface – a List is a Collection of items where the items have a position. (i.e arrays, ArrayLists, LinkedLists, etc.)

d) Implementing an Interface

```
public class InchRuler implements Measurable
{
      private int length_in_inches;

      public int getMeasure()                   // cont forget to make public
      {
            return length_in_inches * CM_PER_INCH;
      }
}
```

e) Using and casting interfaces and classes

```
public class Test
{
      InchRuler ruler = new InchRuler(12);      // a 12-inch ruler
      Measurable x = ruler;                     // ruler can be “upcasted” to Measurable
      …println(x.getMeasure());                 // ok
      …println(x.getUnit());                    // illegal, can only use getMeasure()

      InchRuler ruler2 = (InchRuler) x;         // OK, since KNOW x is an InchRuler
      InchRuler ruler3 = (FootRuler) x;         // illegal unless FootRuler extends InchRuler

      Measurable y = new Measurable();          // illegal (can’t instantiate interace)
}
```

7. Generics Lecture (Java5—jre1.5)

Why:

Generics solves a long-standing problem in Java when a project has multiple programmers over a long period of time.

Example:

When storing items (objects) in an array or ArrayList or any other JFC data structures (Maps, Lists, etc), the data is stored as type "Object". However, other Java programmers may not know what Type you have really stored. As long as YOU know what you are storing and retrieving, then you can be safe when you CAST your data. However, if you are the one responsible for storing but another programmer somewhere else is in charge of retrieving your data, then you have a problem. How does the other programmer know what to cast?

Example using the pre-Java5 way:

```
// This method tries to remove a "4" from an ArrayList called "c".
public void expurgate(ArrayList c)          //we don't know what is in "c"
{
     for (int i = 0; i<c.size(); i++)
       if (((???) c.get(i) == 4)           //what should the programmer cast be?
          i.set(i, null);                  //programmer must guess the Type
}
```

Example using the Java5 Generics (aka parameterized Types):

```
// This method removes 4-letter words from a ArrayList called "c"
public void expurgate(ArrayList<String> c)  // forcing programmer to use Strings
{                                           // which will be checked by Compiler
     Iterator<String> i = c.iterator();     // Iterators are also using Strings
     while (i.hasNext() )                   // (lecture on Iterators is later)
      if (i.next().length() == 4)           // Automatic casting to String
        i.remove();
}
```

Notes:

1. When you see the code <Type>, read it as "of Type"; the declaration reads as "ArrayList of String c."
2. The main advantages are:
   a. Declarations take longer to code but using the variables takes less typing.
   b. Using generics makes code clearer and safer, since compiler checks rather than runtime exception
   c. We have eliminated an unsafe cast and a number of extra parentheses
   d. We have moved part of the specification of the method from a comment to its signature, so the compiler can verify at compile time that the type constraints are not violated at run time.
   e. If the program compiles without warnings, we can state with certainty that it will not throw a ClassCastException at run time.
   f. The net effect of using generics, especially in large programs, is improved readability and robustness.
3. The main disadvantage are:
   a. It makes a bit more programming. We must add <String> to our declarations.
   b. Parameter type information is not available at run time,
   c. Automatically generated casts may fail when interoperating with ill-behaved legacy (old) code

Examples for Using Generics (we won't cover Declaring/Designing Generic classes in APCS)

old way (legacy) — *new Generic way*

```
private ArrayList  animals;          private ArrayList<Animal>     animals;
private Map        environment;      private Map<Location, Animal> environment;
```

E. Statements – Java comes with the following statements

1. Declarations – already covered in previous section

a) Constant – `public static final int PI = 3.14159267;`

b) Variables

c) Class

d) Interface

e) Methods

f) Parameters

2. Expression / Assignments

a) Use of = or other expression (unary, binary, or ternary) followed by a semicolon

3. Flow Control

a) Default is Sequential execution – one line after another

b) Conditional

(1) If statement

```
if (boolean expression) {...}
```

(2) If…else statement

```
if (boolean expression)
      {...}
else
      {...}
```

(3) Switch statement – not tested on AP exam or this course

```
switch (integet expression)
{
      case intValue1: statement;
      case intValue2: statement;
      ...
      default: statement;
}
```

c) Return statement

```
return primitiveValue or reference;
```

d) Throw statement – abruptly terminates current method and resumes control to closest (scope-wise) catch block

```
throws expression;
// expression must evaluate to reference to object of class Throwable
```

e) Break statement

```
break;      // exits out of local block
```

f) Continue statement

```
continue;   // skips to end of block
```

4. Loop / iteration (initialize Loop Control Variable, test LCV, update LCV) **

a) For loop

```
for (init LCV; test LCV; update LCV) {...}
```

b) While loop

```
init LCV;
while (test LCV)
{
       ...;
      update LCV;
}
```

c) Do…while loop – not used on AP test or this course

```
init LCV;
do
{
      update LCV;
       ...;
}
while (test LCV);
```

d) Iterator (interface) – demonstrated later

e) Enhanced For-Loop (Java5—jre1.5) (aka "for each")

Why:

There are times when the "for loop" and "while loop" are more complex than needed, especially for JFC Collections (ArrayLists, Maps, Sets, etc.)

The pre-Java5 "For" looping:

```
// using LCV
int values[] = ....;
int sum;
for (int i=0; i<values.length; i++)
       sume += value;


// iterators
Set names = ...;
Iterator i = names.Iterator();
while (i.hasNext())
   System.out.println((String)i.next());
```

Java5 "For Each" looping:

```
// looping over primitive arrays
int values[] = ...;    // array declaration
int sum;               // total of elements
for (int i : values)   // no more explicit
       sum += value;   // LCV initialize, test,
                       // or updating.

// looping over Collections and using generics
Set<String> names = ...;
for (String n : names)
       System.out.println(n);
```

Notes:

1. Code is easier to read (once you get used to it) and you wont need to scan the loop for index errors
2. Good for "simple" and "common" looping constructs (i.e. may be given on the APCS exam but you won't have to write in your free response part)
3. Don't use if need to do complex or unusual situations (i.e. traversing an array in reverse order).

5. **Method calls**

   a) **Direct calls**

   b) **Recursion or sefl-referencing (often unstable and results in infinite loop)**

   c) **Calls using Generics**

6. **Other statements**

   a) **Block statement (using {…}). Implies another level of scoping rules**

   b) **Import**

   c) **Package**

   d) **Try blocks (i.e. Exception handling) – not test on AP Exam or in class**

F. Special Topics

1. **Exception Handling (to deal with runtime errors to avoid system crashes)**

   a) **Checked – must be "caught"**

   b) **Unchecked**

   c) **Example**

```
try
{
      // statements that might cause a crash
}
catch (ExceptionType e)
{
      // handling of possible problem
      System.out.printl("error: "+e)
}
```

   d) **Common Exceptions to be known for AP exam**

```
NullPointerException;              // object not instantiated
ArrayIndexOutOfBoundsException;    //
ArithmeticException;               // divide by zero
ClassCastException;                // wrong class for downcasting
IllegalArgumentException;          // wrong argument for method call
IllegalStateException;             // JRE not ready (network error)
NoSuchElementException;            // iterator error
```

2. Event Handling – implementing ActionListener interface (i.e. actionPerformed(ActionEvent e) method)

3. Graphics

a) AWT – original graphics, OS dependant (use of Panel, Button, TextField, etc.)

b) Swing – newer graphics, less OS dependant, "lightweight" (use of JPanel, JButton, etc.)

c) User Interfaces

(1) Components – Jbuttons, JTextFields, etc

(2) Containers – Jframes, JPanels, etc. (not Containers extend Components and therefore "are components")

(3) Layouts – GridBagLayout, FlowLayout, etc.

4. Class hierarchy – inheritance (extending)

a) Overloading – same method name but different signatures, can be determined at compile time

b) Overridding – same method signature as parent class, may be determined at runtime

c) Polymorphism – the ability to decide at runtime which method is to be used based upon the object's class at runtime. Also related to casting between levels within the hierarchy. (see getClass() method of Object class)

5. Threads – not on AP test… useful for animation and multiple programs