CISC124 Exam Study Notes

History Of Java

- Language was first developed by James Gosling at Sun Microsystems in 1991
- He was designing a language called "Oak" for "Green Project"
- Green Project envisaged the centralized control of many processor-based devices in home
- "Oak" was designed to be a robust, efficient language with maximum profitability to different processors
- Green Project flopped

- In early 90s, Internet was blossoming. Web pages had to do more than just display static text and graphics
- Needed dynamic and interactive content
- But web pages are viewed on a wide variety of platforms, from Mac's to Unix to IBM-PCs
- So any page-embedded language would need to run on all platforms
- Needed a **robust, compact, and multiplatform** language, so Oak was dusted off and call it something racy, like Java!

- In 1994, Sun demonstrated the use of Java in small bundles of code embedded in a web page called **applets**
- Netscape browsers started supporting applets in 1995, starting Java's rise to fame
- Sun programmers continued to develop a code base for the language, adding many libraries
- They showed that Java could be used for more than just applets, and that full-blown application could be written in this high-level language

Recent History

- Early 2010, Oracle acquired Sun Microsystems
- Oracle seems to be sticking with their promise as demonstrated with recent released of Java 7 and 8
- The second "incarnation" of JavaFX has also been released replacing Swing

How Java Works

- Java standard (the syntax) is identical for all platforms
- A **compiler** (part of JDK sometimes called javac.exe) which is designed to run on your development platform, compiles your **source code** (*.java file) to a **byte code** file (*.cass file)
- The byte code file is **platform-independent** and is the thing you attach to your web page as an applet
- Every browser written or every platform and OS can have an embedded code processor called **JVM** (Java Virtual Machine) built in

- The JVM takes the byte code and executes it by generating the machine code that will be recognized by the platform that is running the browser
- Later, people took the JVM out of browser so they could run stand-alone Java applications. This is the **JRE** (Java Runtime Engine) (java.exe)
- The concept of write once, run anywhere is very appealing

- However, all IDE's must use the appropriate JDK in the background
- Two components of JDK are: **javac.exe and java.exe**
- javac.exe is the byte code compiler and java.exe is the JRE which executes the byte code file

- Compilation is the process of converting the *.java file to a *.class file (the byte code file) this is done by calling javac.exe in the background, and supplying that program with all the required command line parameters

Java.exe Program:

- accepts the byte code file
- links in any required libraries
- creates executable code in memory
- converts it to machine language
- and sends it to the CPU

- The java.exe program must know the right machine language commands for just the type of CPU it is designed for

OOP in Java

- A class or "object definition" or an "object" consists of **instance variables and or methods**
- By convention, instance variables are all declared before the methods
- In Java, a **class** is an **Object**, and an **Object** is a **class**
- Code and attributes cannot be defined outside of a class
- The only code that can exist outside a method are attributes or other ("inner") class definitions

Attributes

- Also called **class variables** or an **instance variables** or **fields**
- Declared within a class at the same level at the method declaration
- These variables are known to all methods within a class (their "scope")
- You can control their privacy and the way they are stored in memory

Access Modifiers

public: means the attribute or method is available to any external class (as well as inside the class)

private: means that the attribute or method, the "member" is only available inside the class which it is declared

protected: means the member is only public to classes in the same package as the class in which the member is declared

static (First Pass)

- static means different things depending on where it is used
- public static members are available outside the class without the need to instantiate the class
- Any static member remains in memory until the program is complete
- Since main is static, it can only invoke other static methods when they the in the same class

Attribute Declaration

Syntax:

[private|public] [static] [final] type attributeName [= literalValue];

- Note that the type part is not optional — this is why java is a "declarative" language
- And a variable cannot change its type later ("static typing")
- You cannot use a variable unless you have declared it first

Variable Declaration

- Declaring a variable inside a method gives that variable the SCOPE of just inside the method, not outside the method
- Generally, a variable is only available inside the block {…} in which it is declared

Method Declaration ("Header")

Syntax:
[private|public] [static] [final] returnType methodName ([parameterList]) {…}

- If main invokes methods or uses attributes in the same class as itself then those attributes and method must also be declared static
- A method must have a returnType
- The returnType can be any single Object or a primitive type
- If the method does not return anything, then the keyword void is used instead
- parameterList provides a means of passing items, or parameters into a method
- it can consist of one or many parameters, separated by commas

Main Method

- For the JVM to run an application, it must know where to start
- By design the starting point is always the execution of the main method
- The JVM expects the main method to be declared exactly as shown — the only thing you can change is the name of the String array, called args above

Methods - the return Statement

- Also important: Unless the return type is void, the method must contain at least one return statement. A void method can also use a return statement without anything after it, as a means of termination of the method
- A method always stops (terminates) when a return statement is encountered

Primitive Types in Java

char byte short int long float double boolean

- Everything else in Java is an Object
- Object construction often involves the **data abstraction** of several primitive type attributes

Integer Primitive Types

- Byte short int long
- For Byte: from -128 to 127 inclusive (1 byte)
- For Short: from -32768 to 32767 inclusive (2 bytes)
- For int: from -21347483648 to 2147483647 inclusive (4 bytes)
- For long: from -9223372036854775808 to 9223372036854775807 inclusive (8 bytes)
- A "byte" is 8 bits, where a "bit" is either a 1 or 0

Number Ranges

- Memory limitations and the system used by Java (two's complement) to store numbers determines the actual numbers
- The Wrapper classes can be used to provide the values

Real Primitive Types
- Also called “Floating Point” Types
- float or double
- For float: (4 bytes) roughly
- For double: (8 bytes)

Character Primitive Type
- char
- From ‘\u0000\ to ‘\uffff’ inclusive that is from 0 to 65535 (base 10) or 0 to ffff (base 16 r hexadecimal)
- A variable of the char type represented a Unicode character, can also be represented as ‘a’ or ‘8’ etc

Boolean Primitive Type
- Boolean is either true or false

String Objects
- String’s are not primitive data types, but are **objects**

Array Declaration
- **new** is always involved with the instantiation of an Object
- Arrays are Objects in Java
- An array literal is just a set of same-type values separated by commas in a set of { }
- To get the size of array use the .length attribute
- Array indices range from 0 to .length - 1
- Cannot dynamically size an array in Java
- In Java, you cannot use pointers to access array elements only the indices
- You can never obtain a memory address in Java
- The first array element is at index 0
- In Java, you will get an immediate error and your program will halt if you try to go beyond the bounds of an array

Literal Values
- Integers for ex: 12 -142 0 333444891
- If you write these kinds of numbers into your program, Java will assume them to be of type “int” and store them accordingly
- If you want them to be type “long” then you must append a “L” to the number

43L 99938475L -22223487L

- Real or “Floating Point” numbers: 4.5 -1.0 3.457E-10
- These literals are assumed to be of “double” type
- If you want them to be stored as “float” types, append an “F”
- Ex. 3.45F

Binary, Octal and Hex Literals
- Use the prefix (0b or 0B) in front of the numbers to get a binary literal
- Octal, just use —0
- Hex use 0x
- You can also use the underscore _ in-between digits to aid in visualizing number
- System.out.println() by default displays the numbers in base 10
- Use printf to show the numbers in another codex or base

Legal Variable Names
- Java names may contain any number of letters, numbers and underscore characters but they must begin with a letter

Standard Java Naming Conventions:
- Names beginning with lowercase letters are variables or methods
- Names beginning with uppercase letters are class names
- Successive words within a name are capitalized (Camel Case)
- Names in capital letters are constants

Variable Declaration:
- Java may prevent you from using variables that are not initialized
- So it is often good practice to initialize your variables before use

Constant Attribute Declaration

Syntax:

[private|public] [static] final type ATTRIBUTE_NAME = literal_value;

- The java keyword **final** can be used to make sure a variable value is no longer “variable”
- Usually they are declared **public static**
- The value must be assigned

- Java will not allow your program to change a constant’s value once it has been declared
- Note that constant names are all in upper-case by convention

Type Casting
- When a value of one type is stored into a variable of another type
- Casting of primitive types in one direction is automatic, you do not have to deliberately or “explicitily” cast
- A value to the left can be assigned to a variable to the right without explicit casting:]

byte > short > int > long > float> double

Casting Operator
- If you really want to cast in the other direction, then you must make an explicit cast

int anotherVar = (int)345.892;

is legal, the “(int)” part of the statement casts the double to an int
- Note how numbers are truncated not rounded!

What Makes an Expression?

What are all the components available to a programmer to use to put a line of code togher?

- variables
- literal values
- keywords
- operators
- method invocations
- Punctuation

Arithmetic Operators

- The standard binary arithmetic operators in Java are:
  Addition, Subtraction, Multiplication, Division, Mod (%)
- All these operators apply to all numeric primitive data types
- All require values on both sides of the operator (why they are called binary operators)
- Java does not have ** or ^ (exponentiation), have to use **Math.pow(x,y)**

Integer Arithmetic

- Arithmetic operators between integers produce integer results by truncating the answer, fractional parts are discarded
- If you have an integer on both sides of an arithmetic operator, the result will be an integer

Mixed Type Arithmetic Expressions

- Suppose you have a "mixed type" expression involving an arithmetic operator
- To evaluate the expression, Java will cast one side to match the other
- For ex: if one side is an int and the other side is a double, the int will be automatically cast to a double before the operation takes place

Strings and the "+" Operator

- Not only can "+" operate on numeric values, but it can also handle String's on either or both sides
- If one side is not a String, it will be changed to one, and then it will be concatenated to the String on the other side:

Ex: 4 + "you" stores as "4you"
"apples" + "oranges" + 9 + 9 stores as "applesoranges99"
3+7+"little piggies" stores as "10 little piggies"

- Expressions are evaluated from left to right unless precedence rules apply

Unary Arithmetic Operators

- Unary operators include "+" and "-" where -aNum negates the value produced by aNum
- They also include the increment (++) and decrement (- -) operators which increase or decease an integer value by 1
- **Pre increment** and **Pre decrement** operators appear **before** a variable. They increment or decrement the value of the variable before it is used in the expression
- **Post increment** and **Post decrement** operators appear after a variable. They increment or decrement the value of the variable after it is used in the expression
- It is better to keep increment and decrement operators out of larger expressions

Assignment Operators
= set equals to
***=** multiply and set equal to
/= divide and set equal to
-= subtract and set equal to
+= add and set equal to

- An assignment statement in Java has form: varName = expression;
- The expression is evaluated, and the result of the expression is stored in the specified variable

Logical Binary Operators
- Return either true or false

== equals to
!= not equals to
> greater than < less than
>= greater than or equal to
<= less than or equal to
&, && logical “And”
|, || logical “Or”

- The only unary logical operator is “!”
- Called the “not” operator
- It reverse the logical value of a boolean

Aside: “|” or “||”
- A single | or & ALWAYS evaluates both sides of the expression whether it is necessary to not
- && stops if the left side is false, || stops if the left side is true

Precedence Rules
- Operator precedence rules determine which operations take place in what order:
  - Unary operators and casting are done first
  - Then *, /, %
  - Then +. -
  - Then >, <, >=, <=
  - Then ==, !=
  - then &, &&, |, ||
  - Then =, *=, /=, -=, +=
- Use “()” to control order of operations as the expression inside ( ) will be evaluated before stuff outside of ()

Expressions
- Expressions are combinations of variables, literal values, operators, keywords, method calls
- What are all the components available to a programmer to use to put a line of code together?
  - variables, literals values, keywords, operators, method invocations, punctuation

Method Invocations
Three Aspects to Consider:

1. Naming the method
2. Providing argument(s) or not
3. Doing something with the return value or not

1. Naming the Method

- if the method is in the same class, then just use the name of the method

(the compiler assumes the existence of the object reference to the current object as this. or it assumes the name of the current object, when static)

- If the method is not the same class you must first identify the object owning the method and obtain the method using the dot operator
- If the method is declared as static, then you can invoke the method without instantiating the class that contains the method

2. Providing Arguments for the Parameters

- If the method has been declared to accept arguments
- if the method does not have any parameters, then you must still use empty brackets when you invoke the method

3. Using a Return Value

- A non-void method will return something
- you can use that "something" in an expression, or just store it in a variable
- The method has declared the type of that "something"
- If the method was declared as void, you will not get a return value and you can only invoke the method by itself, not as part of an expression or an assignment statement

Punctuation - Java Whitespace

- Multiplace spaces are treated as one space
- Leading and trailing spaces are ignored
- Tabs and empty lines (line feeds) are ignored
- Line Continuation:
  - Long lines can be continued on the line below — break the line anywhere there is a space, but not in the middle of a string
  - For good stype, indent the line to show it is a continuation

- Comment: inline //, block /*…*/
- Comma , used in parameter lists and array literals
- Semicolon ; used to end a statement and with for loop syntax
- Colon : used with switch statements and "for each" loop
- Period or "dot operator" . used with objects to obtain members
- Also [ ] , ( ) and { }

Conditionals or "Selection Statements" : if - else statement
Abbreviated if Statement

- Uses the "ternary operator" - ?
- expression1 ? expression2 : expression 3
- Expression1 must evaluate to boolean
- expression2 (when true) and expression 3(when false) must evaluate to same type

Dangling else Problem
- It is not unusual to nest if statements inside each other
- One issue that can arise is the “Dangling else” problem
- Indentation might give a visual clue, but has no syntactic meaning
- The else should be associated with the closest if
- Use { } if necessary

Switch Statement
- The code to be run depends on which val# value matches expression
- If none match, the statements after the default: clause are run
- The expression and val# values (of “Case Labels”) must all be of the same integer (or char) type
- The break; statements make sure that following cases are not executed after a match has been made
- It is possible to do multiple cases on one line, but it is clumsy

Repetition or Using “Loops”
Java has: while, do/while, for, the for each loop

Loops (Misc)
- Don’t declare variables inside loops, as the repeated declaration process uses up time and memory necessarily
- There is not limit in Java to how many levels you can nest loops
- It is customary but not necessary, to use the variables i,j,k as loop counters when the counter has no intrinsic meaning

Other Java Keywords used with Loops
- The **continue** statement interrupts the execution of a loop and returns control to the top of the loop
- The **break** statement interrupts the execution of a loop, and transfers control to the first statement after the loop
- Only use these keywords when it makes your code easier to read
- Avoid the use of more than one break or continue inside a loop
- Overuse of break statements can lead to “spaghetti” code

Building Modular Code
- Easier to build, test, debug, modify and share
- Building objects just takes modular design to the next level

Designing Functions
- Functions are written o avoid repeating code so, make sure you only have one function to do the “one thing”
- Functions should be short: if you can satisfy all the other rules and the code still explains itself, then the function is short enough

Functions should only do one thing and do it well
- Keep all code within the function at the same level of abstraction
- Your program should be readable as a top-down narrative. The most abstract functions will be at the top of the program leading to the least abstract functions further down

- Use less than three parameters whenever possible
- Try to use parameters for input only
- Flag parameters are ugly, they are saying that the function does at least two things
- If needed, multiple parameters can be grouped into an object or list

- The function should not spawn any side effects
  - such as changing the contents of variables passed by reference
  - or spawning off another process that is at the same level of abstraction as function itself
- A function should either do something or answer something, not both

Some Useful Java Classes
- The classes defined in the java.lang package are automatically imported for you
- They include:
  The Wrapper classes, Math, Object, String, System, Thread

static Methods
- static attributes and methods are loaded once into memory and not garbaged collected until main is finished
- These methods will run faster the second time (and later) they are invoked
- Generally they are utility methods that do not depend on the values of a class' attributes
- static methods can be invoked without instantiation of the Object that owns them

- static methods and attributes are shared by all instances of a class, there is only one copy of these methods in memory
- A static method can only invoke other static methods in its own class — you can't have pieces of code disappearing from a static method in memory
- This is all done for reasons of ease of use and efficiency

What Is An Object?

An entity that exists in an operating computer program that has State, behaviour, identity

- The STATE of an Object is the collection of **information held in that object**. This information **may change** over time, as a result of operations carried out on the Object
- The BEHAVIOUR is the **collection of operations** that an Object **supports**
- The IDENTITY of an Object allows the program **access** to a specific Object

- An Object represents **real** or **abstract entities**

- An Object representing a real entity for example: State: awake, hungry, purring. Behaviour: scratchingSofa Identity: ginger

What is a Class?

- You can have many (possibly infinite) different Objects with different values for each State category (or ATTRIBUTE)
- But if each of these Objects has the same set of possible behaviours then you can group these Objects together into a Class

- A class is defined in the source code of a program: It defines…
  - The operations that are allowed on instances of this class (the methods)
  - The possible categories of state that are allowed for instances of this class (the attributes)
- Instances of a class are created when the program is running

- The State of an Object instance can be partly or completely defined when the instance is created
- The State will likely change when operations are carried out on the instance
- However, **attributes cannot be added or removed** and **behaviour cannot be added or removed**. These are **defined in the Class**

Object Categories

In a program, Objects will probably fall into one of these general categories:

- Tangible things (ex. Cat)
- Agents (StringTokenizer)
- Events and transactions (MouseEvent)
- Users and roles (Administrator)
- Systems (MailSystem)
- System interfaces and devices (File)
- Foundational classes (String)

Object Extremes

Two extremes of object structure:

Utility Classes: All STATIC methods and attributes

You do not instantiate these classes, there is not point

The Math class for example

Customizable Classes: All NON-STATIC methods and attributes

Attribute values (some or all) must be set at the time of instantiation before the class can be used

Scanner Class for example

- Many classes fall **in-between** these two extremes:
  - A mix of static and non-static methods
  - static methods have nothing to do with the attributes and so can be used without instantiation of the class
  - Non-static methods depend on the attributes which must be set through instantiation
  - Wrapper classes for example: Double, Integer, etc.

Encapsulation

Is the process of defining a Class that has at least one customizable attribute

- Encapsulation is about the **abstraction** or **containment** of the attributes defined in the class
- In Java, methods and attributes **must** be encapsulated or contained in a class definition

The Best Object Design:

- Supports the re-usability of code
- Provides a well-defined interface to other objects and the user(s)
- Builds into an Object the code that ensures the **integrity** of the data stored in the Object by:
  - Making sure that initial data values are "legal"
  - Controlling (or Preventing) changes to data
  - Preventing "Privacy Leaks" when data is returned out of an Object
- Works well with modular design practices making code easier to write, test, and debug

- This is the driving principle for the design for all Objects in the Java API!
- **Attributes must be declared private, so that the class that owns them can control how they are set**

If attributes are private then how can a user of the class assign the values of the attributes?

Through methods of course!

Constructor(s) (when instantiated only)

Mutator(s) (anytime after instantiation)

- Within these methods, you can write code to check the parameters for validity
- If values are NOT LEGAL, throw an exception!

Defining Exceptions

- You can throw an exception already defined in Java but, Most of the time you will want to throw your own exception objects
- Keywords: extends, super, throw, throws

Assigning Private Attributes - Constructors

- Constructors are special methods that have the same name as the class in which they reside, but have NO RETURN TYPE (not even void)
- They MUST be public
- They are only executed ONCE, when the object is being instantiated. You can't invoke them later
- Constructors are often OVERLOADED, which means you can have MANY constructors with different parameter lists

- Note that once you have written a constructor with parameters, you can no longer use a constructor without any parameters, called the default constructor
- If you want an empty constructor, you must write one yourself, in addition to the parameterized constructor(s)

Suppose you don't want to force the user to supply all parameters when he instantiates the object. How are you going to do this? Overload the constructors

Suppose you want to edit parameters after you have created the object. How are you going to do this? Provide mutator(s)

Preventing Instantiation
- Provide ONLY the DEFAULT CONSTRUCTOR and make it PRIVATE
- Used by math class for example, there is no need to instantiate the Math class since all its methods and attributes are STATIC

Assigning Private Attributes - Mutator Methods
- Called "set" methods — in fact, by convention the word "set" is the first word in the method name
- These methods must also check for legal parameters
- Usually the constructor invokes mutators, when they exist

The "this" Thing
- Used by one constructor to invoke the other one
- Can be used to supply a reference to the current object in that object's code
- this means myself
- Another use, suppose I used the same attribute name as a constructor parameter — year
  In constructor I would need to say: this.year = year;

Accessors
- Accessor methods return the value of an attribute
- By convention, accessor methods start with the word "get"
- One accessor per attribute

Ex.

```
public int getNumMunchkins() {
        return numMunchkins;
}//end getNumMunchkins Accessor
```

- They are pretty simple methods except when you are returning an object

Useful Java Classes

Classes defined in java.lang package are automatically imported as they are used quite often

- The Wrapper classes
- Math
- Object
- String
- System
- Thread

WMOSST

static Methods
- static attributes and methods are loaded once into memory and not garbaged collected until main is FINISHED
- Generally, they are utility methods that do not depend on the values of a class' attributes
- static methods can be invoked WITHOUT instantiation of the Object that owns them
- static methods and attributes are SHARED by ALL instances of a class - There is only ONE copy of these methods in memory
- static method can only invoke other static methods in its own class, can't have pieces of code disappearing from a static method in memory

Math Class
- A collection of **static constants** and **static mathematical methods**
- you cannot instantiate the Math class

Wrapper Classes
- Sometimes its necessary for a PRIMITIVE TYPE value to be an OBJECT, rather than just a primitive type
  - Some data structures only store Objects
  - Some Java methods only work on Objects
- Each primitive type has an associated Wrapper class:

char	Character
int	Integer
long	Long
float	Float
double	Double

- Each wrapper class Object can hold the value that would normally be contained in the primitive type variable, but now has a number of useful static methods

Ex. Integer number = new Integer(46); //wrapping
int aNumber = number.intValue(); //aNumber is 46

- The Character wrapper class: has methods to convert between ASCII and Unicode numeric values and characters
  - isDigit(character) returns true if character is a digit
  - isLetter(character)
  - isUpperCase(character)
  - isLowerCase(character)
  - isWhitespace(character)
  - toLowerCase()
  - toUpperCase()

System Class
- Have used System.out.println(), print(), printf(), and err.println()

Other Useful System Class Methods:
System.currentTimeMillis()
Returns, as a long, the number of milliseconds elapsed since midnight 01/01/1970
System.exit(0)
Immediate termination of program

System.getProperties()
    All kinds of system specific info
System.getProperty(string)
    Displays single system property
System.nanoTime()
    Time in nanoseconds

String Class
- Since Strings are Objects they can have methods
- There are 67 String methods
    - length()
    - equals(OtherString)
    - equalsIgnoreCase(OtherString)
    - toLowerCase()
    - toUpperCase()
    - trim()
    - charAt(Position)
    - substring(Start)
    - substring(Start, End)
    - IndexOf(SearchString)
    - replace(oldChar, newChar)
    - startsWith(PrefixString)
    - endsWith(SuffixString)
    - valueOf(integer)
- String's DO NOT HAVE ATTRIBUTES

- Constructors are special methods with the same name as the class that are invoked when you instantiate the class
- The String class has 15 of these — the constructor is "overloaded"

- Strings are **immutable** meaning that they cannot be altered, only re-assigned
- There are no methods that can alter characters inside a string while leaving the rest alone
- Arrays are **mutable** in contrast, any element can be changed

Mutability
- For an array you can use [ ] on either side of an assignment operator
- A String cannot use [ ] to access individual characters, only the .charAt() method
- The .charAt() method cannot be used on left hand side of assignment operator

Other java.lang Classes
- Object is the base class for all objects in java
- Thread is a base class used to create threads in multi-threaded program.
- A class not in java.lang:

    String Tokenizer Class

    - This useful class is in the "java.util" package, so you need to import java.util.*
    - This class provides an easy way of parsing strings up into pieces, called "tokens"
    - Tokens are separated by "delimiters" that you can specify or you can accept a list of default delimiters
    - The constructor method for this class is overloaded

- So, when you create an Object of type StringTokenizer, you have 3 options
    new StringTokenizer(String s)
    new StringTokenizer(String s, String delim)
    new StringTokenizer(String s, String delim, boolean returnTokens)
- s is the String you want to "tokenize"
- delim is a list of delimiters, by default it is: \t\n\r
- You can specify your own list of delimiters if you provide a different String for the second parameter
- If you specify a **true** for the final parameter, then delimiters will also be provided as tokens
- Default is false, delimiters are not provided as tokens
- Note that the StringTokenizer object is emptied out as tokens are removed from it
- You will need to re-create the object in order to tokenize it again

Method Overloading

- A method can have the same name in many different classes
- "Overloading" is when a **method name is used more than once** in method declarations with the **same class**
- The rule is that no two methods with the same name within a class can have the same number and/or types of parameters in the method declarations (the "NOT" rule)
- Its for convenience!
- Java does not have default arguments
- Allows the user to call a method without requiring him to supply values for all the parameters
- One method name can be used with many different types and combinations of parameters
- Allows the programmer to keep an old method definition in a class for "backwards compatibility"

How does it work?

- Java looks through all methods until the parameter types match with the list of arguments supplied by the user. If none match, Java tries to cast types in order to get a match. (Only "widening" casting like int to double)

- You can have as many overloaded method definitions as you want, as long as they are differentiated by the type and/or number of the parameters listed in the definition
- **Do not change the return type, its tacky!!**

Exceptions

How can a method indicate that it is unable to return what it is supposed to return?
How can a method deliver details about the error condition?
How else can you prevent the instantiation of an Object?

- The limitation of only returning a single "thing" means that you either designate error values for the "thing" or you have some other way to return the indication of an error
- The designers of Java followed conventions used by many other OOP languages — they allowed for another way to get something out of a method. However, an **exception is thrown, not returned**

- **Exceptions are Objects**

- When an error condition is encountered, a method can throw an instance of a pre-defined exception Object
- A method can throw several exceptions, one for each possible kind of error condition

Exceptions: Propagation
- If a method throws an exception, then that method is immediately halted and there is no need for any return value, even if the method is non-void

- The invoking method then receives the exception, if it does not catch it, then it goes to the next invoking method — all the way to main, if necessary (Called cascading or propagation)
- Finally, if main does not catch the exception, your program crashes and a message is sent tot the console window

Exceptions: Message Handler
- How does an Exception Object contain information about the error condition?
  - The type of the Object:
    - IOException
    - NumberFormatException
    - FileNotFoundException
    - ArrayIndexOutOfBoundsException etc…
- So the method should throw a relevant exception object
- Turns out that exceptions can also carry a String message

- An exception is just a “Bearer of Bad Tidings”
- The exception itself cannot do anything about the problem, but it can act to stop your program

Exceptions: Catching
- This is done with a “try/catch” block
- The compiler will force you to use try/catch block when you invoke methods that throw exceptions

Syntax:

```
try{
        //block of statements that might generate an exception
} catch (exception_type identifier) {
        //block of statements
} [ catch (exception_type identifier) {
        //block of statements …
} ] [ finally {
        // block of statements
}]
```

- You **must have at least one catch block** after the “try block” (otherwise the try block would be useless!)
- You can have many catch blocks, one for each exception you are trying to catch
- The code in the “finally” block is ALWAYS executed, whether an exception is thrown, caught, or not

Checked vs. Unchecked Exceptions

- Checked exceptions must either be **caught in a method** or **thrown from that method** (using the throws clause in the method header)
- You should not catch an un-checked exception or an error
- The compiler will ensure that checked exceptions are handled properly
- Unchecked exceptions occur only at runtime and the compiler does not care about them
- Unchecked exceptions are all sub-classes of java.lang.Error

Try with Resources

- This new syntax is very useful with Java 7&8's improved file I/O syntax

Syntax:

```
try (instantiation; instantiation; …) {
        //other statements that might generate an exception
} [ catch (exception_type identifier) {
        //block of statements
} ] [ catch (exception_type identifier) {
        //block of statements …
} ] [ finally {
        //block of statements
} ]
```

- The instantiation(s) inside the set of ( ) immediately after the try keyword are declared resources that must be local to the try/catch block
- Note that there is no ; at the end of the list and that there does not have to be any catch blocks
- These resources must all implement the Autocloseable interface, which means that the try block can close the resource when it is finished
- Resources will be closed whether or not an exception is thrown because their scope is forced to be in the try block only
- As a result, all use of the resource must also take place in the try block

Scanner Class Tokenizer

- Scanner class has tokenizer built into it
- Scanner uses a regular expression (regex) instead of the (easier to understand, but less powerful) delimiter list
- default of regex is "\p{javaWhitespace} +" which means any number of whitespace characters
- A **whitespace character** is a space, tab, linefeed, formfeed, or carriage return

Aliasing Objects: Array Example

```
int[ ] first = {1,2,3,4,5}'
int[ ] second = {10,20,30,40,50,60,70};
```

(the two arrays will point to different addresses)

```
second = first; //aliasing
```

(the pointer to the first array address is then pointed to the second, where after garbage collection, all the information in second is REPLACED with first)

```
first[4] = 500; //second[4] is also 500
```

- So setting one array to equal another as in: array1 = array2; sets array1 to point to same data memory location that was (and still is) pointed to by array2
- Now changing the value of an element in array2, will change that same element in array1, or vice versa. This makes sense since both array Objects point to the same set of data values in memory!

Garbage Collection
- Some computer programming languages require you to indicate when you are done with variables so the memory they are occupying can be released back to the OS
- Fortunately, Java has an AUTOMATIC garbage collection system:

- variables are garbage collected one you **move outside their scope**
- Object contents are garbage collected when there are **no pointers pointing to the contents**

Aliasing Objects
- Passing an Object into a method results in the method's parameters being aliased to the Object passed (called Passing by Reference!)

Passing Parameters by Reference
Ex. in main:
```
int[ ] arrayA = {1,2,3,4,5};
passArray(arrayA); //invoke passArray
```

The passArray method:
```
public static void passArray(int[ ] arrayB) {
        // arrayB is aliased to arrayB from main
        // making elemental changes to arrayB will also change elements in arrayA in main
        arrayB[3] = 400;
}//end passArray
//arrayA[3] is 400 in main
```

The Rule for parameter passing into methods is:
Objects are passed by reference, primitive types are passed by value

- Only element by element changes in arrays will "stick"
- Re-assigning the array to a pointer that has local scope in a method will not "stick"
- If you make element by element changes using an aliased local pointer (like the parameter), changes **will** "stick"
- Strings are **immutable** so this does not apply. You cannot make elemental changes in a String, even though a String is passed by reference
- So, **mutable** Objects (like arrays) can be passed **into and out** of a method through the parameter list. If a method changes the contents of a mutable Object passed into it, those changes "stick" even when the method is complete

Comparing Objects
Testing arrays and Objects for equality (with the "==" boolean operator is interesting:
- This test will only give true when **both objects** have been **aliased**, using the assignment operator "="

- So even if both arrays have identical contents, "==" will return false, unless both arrays point to the same location
- This means that **comparing Objects** with **"==" only compares pointers not contents**

Standard Encapsulated Structure
- private attributes
- Constructor(s) could be overloaded
- Mutators
- Avoiding privacy leaks
- Constructors and mutators throw exceptions to prevent the creation of an illegal object
- Accessors
- Other standard members: toString, compareTo, equals, clone

Standard Methods — toString(), equals(), compareTo(), clone()

toString() Method
- If we try to print an Object, we will get a string composed of the object type and its has code (Halloween2@923e30)
- So to get more useful view of the contents of the Object, defined a toString() method that returns a String
- This method will be called automatically whenever a String version of the object is need
- It overrides the toString() method from the parent Object class

equals() Method
- Accepts another object of type "Halloween4" and returns true if they are equal, false otherwise
- Usually all the attributes have to match
- Two ways to write an equals() method but both will return true or false
  - Accepts a Halloween4 object as a parameter
  - **Accept an Object as a parameter (Object is the base class for all objects in Java!)**
- The **PROPER way is the second one** — then you will overwrite the equals() method inherited from the base Object class
- The instance keyword checks if the supplied Object is indeed a Halloween4 object before it attempts to cast it (otherwise you get a ClassCastException)
- Then the object is cast to be of type Halloween4 using the casting operator
- Note how the method can refer directly to the attributes of the other object

compareTo() Method
- Compares a supplied object with the current one, based on your comparison criteria
- It returns an int value
- (like the compareTo() method in the String class)
- You have to decide on the basis for comparison
- The base Object class does not have this method, so you don't have to worry about writing one to take an Object as a parameter

clone() Method
- Suppose you want to make a completely separate copy of an int array called stuff:

```
int[ ] clonedStuff = stuff.clone();
```

OR

```
int[ ] clonedStuff = new int[stuff.length];
for (int i = 0; i < stuff.length; i++)
        clonedStuff[i] = stuff[i];
```

- Returns a deep copy of the current object
- A "deep copy" is not aliased in any way to the original object, but contains the same values for each attribute

Annotations

- The "@Override" thingy
- You don't have to use these, but the pre-compiler likes them
- This one tells the pre-compiler that "we know we are overriding toString in the parent class and we are ok with it!

CISC124 Quiz 3 Notes

JavaDoc

- Javadoc.exe is a program that comes with the JDK, its NOT included in the JRE only
- If I have written the class "MyClass.java", that contains properly formatted comments, then running "javadoc MyClass.java" **generates a file "MyClass.html"**
- The html file contains external documentation generated from the formatted comments in the source code
- Normal block comments are /* */ JAVADOC comments are: /** */
- General form of Javadoc tag is @tagName comment
- The tags you use depend on what you are describing (class, method, attribute)
- In the car of methods, you can have a tag for each parameter, return value, and a tag for each thrown exception
- Typically you will only write javadoc comments for public attributes and methods

- Html tags can also be added to the comments to add more formatting to the resulting document:
  - for emphasis
  - <code> for code font
  - <img> for images
  -   for bulleted lists

- The output from Javadoc looks exactly like the API documentation
- The advantage is that when source code is changed, the Javadoc comments can be changed in the source at the same time
- The external documentation is than easily re-generated
- Javadoc also provides a consistent look and feel to these API documents

- Most modern IDE's allow you to run Javadoc from within the environment, and provide tools and wizards to help you create comments

Debugging

- With the debugger, you can run to a breakpoint, stop your program and then execute one line at a time while watching the call stack, variables, and custom expressions

Take a look at the UseDebugger.java file

Testing

- JUnit is a framework designed for this kind of work and it is very easy to use

JUnit Testing:

-

Assertions

Use (expected, actual) or (String, expected, actual):

- assetArrayEqual()
- assertEquals()
- assertFalse()
- assertNotEquals()
- assertNotNull()
- assertNotSame() //for objects

- assertSame() //for objects
- assertNull()
- assertTrue()

assertThat()
- Takes a Matcher<T> object for its second (or third) parameter
- Note that the order is different, the *actual* comes first followed by the *expected*
- You can still have a String message as the first parameter
- You can build more sophisticated assertions with this method
- (tests to see if exceptions are thrown by using: @test (expected=IllegaHalloweenException.class)

No try/catch
- If you are testing code that has a throws declaration, then just have your unit test method throw Exception to satisfy the compiler
- If an exception is thrown unexpectedly (a run-time error) you will get a different kind of failure notice in the report. (NOTE: the testing DOES NOT stop)

Test Suites
- You can combine separate JUnit testing classes into a single suite and run them all at once

Advantages
- Tests are all separate from the code being harnessed
- Very easy to add tests
- Running tests is a very simple and fast
- Results are nicely summarized and you can zoom in on failed tests easily and get some decent diagnostics

Ex. Test can be run on remote classes on a server

Designing and Running Unit Tests
- Start by testing the most concrete independent methods first
- Then test the methods that depend on these ones
- Then test classes
- Then test systems

- You will need to write stub and driver code as required in the testing classes
- Stub code **substitutes** something "fake" to a method that depends upon it
- Driver code **simulates** code that uses the method being tested
- Keep the stub and driver code SIMPLE so that error can only come from the harnessed code, not the testing code

- Start with tests that easily fall into the normal, expected range of inputs. Few or many? Use test generator tools?
- Create more tests for inputs that you suspect are related to each other
- Choose tests for inputs that are just barely legal
- Choose tests for inputs that are way out there!
- Make sure you exercise all the code being harnessed

- Keep all tests unless you have deleted some of the methods/classes that you were testing. You can even leave these in with a "fail" call

- Fixes applied to some methods can invoke a ripple effect, causing previously passed tests to fail
- In a team effort always run (and pass) all of your unit tests before committing your code to the repository

Unit Testing
- You can never test all possible input conditions
- Unit testing doesn't prove that your code is without errors, but you can end up feeling pretty good about your code if it passes all tests

Test Driven Development (TDD)
- How much code can you "implement" without testing it?
- You should NOT separate the process of implementation and testing
- Not surprising — this is the basic tenant of Agile Development
- Is all about writing tests while you are writing, or decomposing your code
- In fact the testing DRIVES and LEADS the coding
- We need a systematic technique to create tests that ensures coverage:

The Three Laws of TDD
1. You may not write production code until you have written a failing unit test
2. You may not write more of a unit test than what is sufficient to cause a failure, and not compiling is a failure
3. You may not write more production code than what is sufficient to pass the current failing test

- So tests should be written at the same time as the production code. You are always one test ahead of what the production code can pass.
- As the amount of production code grows so do the number of tests
- Testing code should be in separate files from production code
- It really helps to have some kind of framework to help organize and run these tests

Packages
- "package" is a Java keyword
- It provides a means of grouping classes together into a package
- All classes in a package share some common theme
- It is used as in: package packageName;
- This line at the top of a class definition, before the public class

- Eclipse prefers you to create classes in a package
- When you create a new Class, specify what package you want it to belong to. This will automatically add the package packageName; line to the top of the class, the proper folder is created in src, and the new class is saved in the folder
- Eventually you will create a *jar (or *.zip) file with all the *.class files in your package
- Put this user library somewhere in or below your **class path** and then another class will be able to import it

- The structure is:
  classpath\folder\packagename
- "folder" can be a series of folders
- The import statement looks like:
  import folder.packagename.*;
- The ".*" says to import all classes in the package, or you can import specific classes

- We have been importing existing packages already
  import java.util.Scanner;
  import java.io.*;
- NOTE: Java automatically imports the java.lang.* package for you
- This package contains many classes fundamental to the Java language:
  - String
  - Math
  - all Wrapper classes (Boolean, Character, Integer, Long, Float, and Double)
  - System, and a few others…

static Import
- Used to import all static methods in a class, so that they can be used as if they were declared within the class that uses the,

Ex. import static java.lang.Math.*;

Protected Access
- So far we have used public and private access modifiers
- Protected is what you get if you don't specify an access modifier
- This means "public inside the package, private outside the package"
- Used in the API
- Use sparingly!

Enumerated Types
- enum is a keyword that is new since Java 5.0 but C has had enum's for a while

Ex. enum IceCream {CHOCOLATE, VANILLA, STRAWBERRY, GOLD_MEDAL_RIBBION, WOLF_PAWS, BUBBLE_GUM};

IceCream favourite = IceCream.GOLD_MEDAL_RIBBON;
IceCream leastFavourite = IceCream.BUBBLE_GUM;

System.out.println("Favourite is " + favourite);
System.out.println("Least Favourite is " + leaveFavourite);
DISPLAYS: Favourite is GOLD_MEDAL_RIBBON
Least Favourite is BUBBLE_GUM

- The contents of IceCream are NOT Strings, but they are a kind of Object
- IceCream is also an Object
- They behave as named constants, so by convention, they should be in upper case
- Once you have specified the possible values for IceCream in the enum declaration, you cannot change them
- A variable of type IceCream can only be assigned to one of the possible types

- The Objects in an enum Object inherit a few methods including:
  - equals(), toString(), compareTo(), ordinal(), values()
- you can use equals() or == to test for equality
  Ex. System.out.println(leastFavourite == IceCream.BUBBLE_GUM);
  prints out true to the console
- compareTo() compares the members on the basis of their position in the enum collection

ordinal() Method
- Returns the numeric location of the value in the list (numbering starts from zero)

Ex.
System.out.println("Location = " + favourite.ordinal());
DISPLAYS: Location = 3

values() Method
- Returns an array of the enum's Objects in exactly the same order

Ex. IceCream[ ] flavours = IceCream.values();
System.out.println(flavours[3]);
DISPLAYS: GOLD_MEDAL_RIBBON

Enumerated Types Summary
What's the point of these things?
- enums are used when you need a short, simple, list of things, or "named values"
- The list has a finite size, will not grow and will not (and cannot) change, unlike arrays
- You want to know that you can access ONLY the defined members of the list and no others
- Modern IDE's will provide a list of all the values, when you type a period after the names of the enum

Yes there are other ways to do this, but an enum is pretty easy to use!

Inner Classes
- Simply defined as a class defined within a class
- The inner class can easily get at all the attributes and methods of the outer class
- The outer class can also access the attributes and methods of the **private inner class** after it **instantiates** the inner class
- However, all private classes, methods, and attributes are hidden outside the outer class

Inner Classes: Why Private?
- If you are declaring a non-static object inside your class, you are only doing so because you want to hide it away
- The object can only be used inside the outer class and is hidden everywhere else

So, whats the point of private inner classes?
(Used quite often with GUI coding)
One Reason, when you wish to use a small class (in terms of the size of its definition), but don't want to bother creating a separate class definition file

- An inner class can be a "hidden" object!
- This is often used with Linked List definitions to define the node object
- Yes, you can have an inner class inside an inner class

public static Inner Classes

What is the point of declaring an inner class public static?

- It would allow you to "categorize" methods into topical groups. Invoke them like:

First:

```
TestClass tc = new TestClass();
```

Then, invoking methods from public static inner classes:

```
tc.Group1.method();
tc.Group2.anotherMethod();
```

Anonymous Classes

- Are a kind of inner class
- Defined by putting the code for the class right in the instantiation statement for the object

```
public class AnonymousClassDemo {
    public static void main(String[] args) {
        MessageSender ms = new MessageSender() {
            public void sendGreeting(String name) {
                System.out.println("Hello " + name" + "!");
            }//end sendGreeting
        }; //end ms, NOTE ";"!
        ms.sendGreeting("Alan");
    }//end main
}//end AnonymousClassDemo
```

- MessageSender is an interface, not an Object
- So you have not actually named the Object that contains the definition of the method sendGreeting()
- ms is an Anonymous Object

- Some coders will tell you that anonymous classes are just classes written by lazy coders
- This is TRUE, but you will see these little beasties used with GUI coding
- Java 9 now has a very tidy solution to using messy anonymous classes in a GUI program — using Lambda Functions

abstract Classes

- It is not unusual to declare a class in the root of a hierarchy to be abstract:

```
public abstract MyClass…
```

- Any class declared this way CANNOT be instantiated
- It can only be extended
- Unlike an interface, an abstract class can also contain concrete method definitions and any kind of attribute
- If a class has one or more abstract methods, the class must be declared abstract as well
- abstract methods have no code in them

Ex. public abstract String getListing();

- A class that extends an abstract class MUST override all the abstract methods in the class, unless it wants to be abstract too
- Unlike the interface, you should write public abstract for each method signature

Why Bother?
- An abstract class forces sub-classes to define certain methods. The shells ensure that the hierarchy has a consistent design
- Also, when declaring a method in a very abstract class, then you don't have to worry about what to do in the method body, especially if it must return a value
- Provides the mechanism for polymorphism!

Interfaces
- Interfaces contain constant attributes and/or abstract methods ONLY
- Abstract methods consist of just the method header — there is a semi-colon instead of { }
- (Abstract classes can contain both concrete and abstract methods, along with any kind of attribute)

- An interface is a design specification, that lists a set of expectations for classes that implement the interface
- Interfaces DO NOT extend Object
- Interfaces can extend multiple interfaces (but not other classes)
- Classes can implement one or many interfaces:
  public class Test implements interface1, interface2, interface3, … { }
- An interface cannot be instantiated, but you can use the interface type as if it is a class
- The interface acts as a "stand-in" for the concrete object that will replace it when the program is running
- The interface "guarantees" the required behaviour of the object it is standing in for

- Attributes in interfaces can only be public final static, you CANNOT use private. You don't even have to specify public final static as it is ASSUMED. Constants must be initialized
- Method signatures must also be public and you don't have to state it explicitly. They are also "abstract", but you do not use the abstract keyword in an interface (considered tacky)
- A class that implements an interface must have a concrete implementation of every method signature in the interface

When designing an interface, you must be careful to use unique method and constant names, in case your interface becomes part of a multiple implementation

Viewing Java Source Code …

The Comparable Interface in Java
Without all the javadoc comments:

```
package java.lang;
import java.util.*;
public interface Comparable<T> {
        public int compareTo( T o);
}
```

- Thats it!, This is a GENERIC interface now
- A class that implements Comparable can be sorted with:
  Array.sort() or Collections.sort() or ArrayList.sort()
- The first two methods are already contained in java (package java.util) and use very fast merge sort or Quicksort algorithms

Why is this implementation necessary?
Implementation in Arrays.mergesort()
The line code from mergeSort() method in java.util.Arrays class:
if (((Comparable) src[mid-1]).compareTo(src[mid]) <= 0)
- This method sorts src[ ] which is of TYPE Object[ ]
- So provided your Objects implements Comparable, this method can sort it!

Interfaces…
- While interfaces are not proper objects in Java, you can "pretend" that they are in some circumstances

Ex. A class can use the interface as an object type, and write code as if you are invoking a method declared in the interface. You can pretend that the specification is an objet
- Part of polymorphism
- An instance of a class that implements an interface can be cast to that interface type

Inheritance
- An OOP design technique that allows you to add to, modify, replace, or simply adopt the methods and attributes in an existing class
- The class that you are modifying is called the **parent** or **super** class, and the resulting class is called the **child** or **sub** class
- The child or sub-class **inherits** all the PUBLIC attributes and methods of the parent or superclass
- The child class **extends** the parent class
- The sub-class is always **more concrete** or **less abstract** than the super class

Why Bother?
- One reason is **code re-use**
- It is also a design tool that allows the coder to more easily model a real-life structure
- You can also use inheritance to enforce code structure on child objects (using interfaces and abstract classes)
- And it is part of the polymorphism mechanism

- Child Objects always have an "is a " relationship with the Parent Object
- In the Person class you would code all the attributes (age, gender, hair colour etc) that describe a person
- The sub classes only need to hold those attributes that are specific to them (such as numExamsToCreate in the Professor Object)
- The sub-classes do not have to repeat any of the code in the super-class

Some Goals of Hierarchy Design
- All attributes in an instance should be defined with meaningful values
- Minimize code repetition
- The same attribute should not be declared in more than one class

- A child class should be more concrete than its parent class
- Design for polymorphism:
    - Use abstraction (abstract classes and interfaces) to ensure common behaviour and to allow for polymorphism
- Control and minimize the public interface of all classes
- Do not have any "pass through" classes that do not contribute to anything
- Make sure the structure is extensible
- Use a consistent variable and method naming scheme throughout the hierarchy
- Follow the "real world" hierarchy as closely as possible

The Object class
- So we are inheriting all the public members of this class — whether we want this stuff or not

extends Keyword
- Creating the Person class:
    public class Person { … }
- Which is the same as:
    public class Person extends Object { … }
- But the "extends Object" is always there, so you don't have to write it
- Creating the Professor and Student classes:
    public class Professor extends Person { … }
    public class Student extends Person { … }

super Keyword
- Provides a reference to the immediate parent class
- NOTE: the compiler will force you to invoke super in a child class' constructor and it must be the first line in the constructor

Polymorphism

Is when a pointer of a parent class ends up pointing to a different child class objects at runtime. Also called "DYNAMIC BINDING" the process also must satisfy early binding
Early Binding: is satisfied when the parent class also owns the method that will end up being invoked from the morphed child class objects. The use of interfaces and abstract classes can make it easier to code for early binding

- Pointers of type Person are pointing to objects of type Student and Professor
- At run time the pointers will morph into their actual object types — this is late binding or dynamic binding

Methods in Sub-Classes
- You have 5 ways to have methods in sub-class:

1. Write a new method (concrete or abstract)
2. **Inherit** the method — it is available to use inside the sub-class or it can be invoked from an instance of the sub-class
3. **Override** the method — You must declare the method with exactly the same method declaration line, the same **signature**, as was used in the super-class
4. **Overload** the method in the super-class
5. **Refine** the method by invoking the super class's method in an overridden method

Inheriting Methods

- As we have discussed, all public methods from the parent class are inherited by the child
- Of course, the parent class may have inherited a bunch of methods from its parent class, so every public method in the hierarchy above the current class is available
- They behave just as if they were written in the child class
- Note that an abstract sub-class can simply inherit an abstract method from an abstract super-class or an interface — it does not have to implement the method. In this case, the sub-class will have to be abstract as well
- suppose you don't want all these methods — How can you avoid inheriting some but not others?

Method Overriding & Refining

- When a sub-class overrides a method of the super-class, then it is the sub-class version that is invoked when called from an instance of the sub-class
- Inside the sub-class, if you wish to invoke the super class version of the overridden method, use the super word
- If the overriding sub-class method invokes the super-class version of the same method, then it has **refined** the super-class method, as well as overriding it

the Use of the final Modifier

- If "final" is included in a class header line, as in:
  public final class ClassName { … }
- Then the class cannot be extended
- If you add "final" to a method definition as in:
  public final void methodName() { … }
- The method cannot be overridden

the Effect of "private"

- private methods and attributes are not inherited by the sub-class
- But, you do inherit public methods and attributes
- You can invoke public methods inherited from the parent class even when they themselves refer to attributes and methods that are private in the parent class

Overloading Methods

- If you use the same method name and return type as in the parent class, but a different parameter list in the sub-class then you are just overloading the inherited method

Multiple Inheritance

- This is when a sub-class extends more than one super-class
- In our diagrams, a sub-class would point to more than one super-class
- Java does not support multiple inheritance (C++ allows it)
- Designers of Java felt that multiple inheritance would lead to structures that are way too complex
- Java designers have supplied the use of interfaces as a way of getting around this restriction
- A sneaky way to get around the lack of multiple inheritance in Java is to combine two classes into one, by making one of them an Inner Class, and then you extend the outer class

Polymorphism, Second Pass
- Where one type can appear as itself, but ends up being used as another type
- Java is a very strongly typed language! Even so, you can allow one object to appear to be something else
- Polymorphism myst be constructed through object extension or interface implementation

Polymorphism: Late and Early Binding
- Your program must always satisfy early binding for it to compile
- Late Binding occurs when the program is running, and only occurs when a variable of a parent-type object is pointing to a child object
- The compiler does not (and cannot) check the late binding to see if it works. A failure of late binding results in a runtime error, not a compiler error

Visualization of Hierarchies
- The easiest way is to view the structure is as a Unified Modelling Language Class Diagram
- This is what we have been doing, but normally more detail is shown

The ArrayList<T> Class
- This is a **generic** data structure class
- The ArrayList<T> class resides in the java.util package. Stores and returns Objects of type T
- You can use the class without specifying a size

Syntax:

```
ArrayList<type> listName = new ArrayList<type>();
```

- type must be an Object, it cannot be a primitive type but can be an array type
- You can supply an initial size to the constructor, if you wish
- In newer versions of Java you do not have to specify the type twice

```
ArrayList<type> listName = new ArrayList< > ();
```

- The empty < > is called the “diamond”
- This is an aspect of what is called **“type inference”**

- Add elements to the collection: myList.add(new Double(456.78)); OR
  myList.add(456.78); //Automatic Boxing
- To get the size of the collection, invoke the size() method:

```
int myListSize = myList.size();
```

- The get(index) method returns the Object at the given index. Note that index positions are numbered from zero
- The set(position, newValue) method changes the Object at the given position
- To insert an element, invoke add(position, newValue). All elements are moved up, and the new value is added at the given position
- The method, remove(position) removes the element at the provided position

- If you are done adding elements to the ArrayList, invoke **trimToSize()** to remove empty element locations
- Also **clone()** DOES NOT WORK properly because it does not return an independent, un-aliased copy. The objects in the collected will still be aliased.

- toArray(tArray) returns the underlying array of type T[ ]. Supply tArray which is an instantiated array of type T. The size will be increased if necessary, so just use a size of zero

Ex. If T is String then invoke as in:

```
arrayList.toArray(new String[0])
```

- This method returns the array String[ ] of the exact size needed to hold all the element in the arrayList

- Once you have accessed an element of an ArrayList<T>, you do not have to cast it back to type T, it is already of the correct type

Ex. To get the number back out of the first element in myList:

```
double aVal = myList.get(0);
```

- Uses **automatic boxing** as well
- This is an advantage over using a collection like Object[ ] (for example)

Your Own Generic Classes

- Also called **"Parameterized Classes"** — kind of like having a parameter in the class header, except it only specifies a type and does not pass a reference, like a parameter does

Ex.

```
public class Sample<T> {
        private T data;
        public void setData(T newData) {
                data = newData.clone();
        }
        public T getData() {
                return data;
        }
}//end sample<T>
```

(Review examples in notes)

Generic Classes: Limitations for T

- You can use an array type, so "<int[ ]>" would be legal
- You cannot specify primitive types, so "<int>" would not be legal
- Neither can you use Exception classes
- You CAN use interface and abstract types

Your Own Generic Classes: Constructor

A construct for Sample<T> class:

```
        public Sample (T newData) {
                data = newData.clone();
        }
```

NOTE: that there is not use of <T> in the constructor

Using the constructor and automatic boxing:

```
        Sample<Double> num3 = new Sample< >(56.7);
        System.out.println(num3.getData());
```

- You can specify any number of type parameters:
  public class Sample2<T1, T2, T3> {…
- You can also "bound" the parameters by specifying a root class:
  public class Sample3<T extends RootClass>
- So only classes that extend RootClass can be used for T (in this example)

Generics and Backwards Compatibility
- Generic code can make for very versatile programs that can be applied to a range of objects — saving you a great deal of boring coding work
- But generic code will not work easily with pre Java 5.0 Code
- and of course, the Java 7 type inferences stunts will not work with pre-Java 7 code
- Generics are a bit like templates in C++ and C#, but are implemented in a different way

Wildcards
- The ?? within < >
- A wildcard can provide a super type generic class for other generic classes
- You can bind a wildcard to classes that extend other classes using the extends keyword

Ex.
  ArrayList<? extends Person? db
- db can be assigned to an object of type ArrayList<Person>, ArrayList<Student> or ArrayList<Professor>
- However, a generic typed with a wildcard is not mutable
- You can get around this using generic methods without the wildcard BUT:
  - This is more restrictive and potentially "brittle"
  - Avoid casting using the generic type
  - This may lead to "Unchecked casts" which reduces the type safety of the generic method
  - Use of Class<T> may be a way around this problem
  - ArrayList<?>[ ] is the only possible type that allows the creation of an array of generic classes
  - But why would you want to? —> easier to create an ArrayList of ArrayList for example

The Class<T> Object
- You can pass a type as an argument into a method using a Class<T> object
- The Object class contains a method .getClass() that returns a Class<T> object. Or you can just use the .class literal on a type

Ex. String aString = "hello!";
System.out.println(aString.getClass());
System.out.println(String.class);
SHOWS: class java.lang.String
  class java.lang.String

- Also has a method called **.newInstance()** that returns an instance of type T
- This method can only invoke the default constructor of the type T. So you must use mutators to set attributes. But now you have an instance to work with!
- In a generic method, the type "T" cannot be instantiated

A Lambda Function

- Kind of like an "anonymous method"

Syntax

- parameters for the method first, no parameters, use { }, one parameter by itself — no brackets required, more than one use (a, b, etc) No types required
- Then the fancy -> "arrow"
- Then the method body. Put more than one statement inside { }

- Can even define an inner interface

Ex.

```
public class LambdaDemo {
    interface MessageSender {
        bond sendGreeting(String aName);
    }//end MessageSender Interface
    public static void main(String[ ] args) {
        MessageSender ms = name -> System.out.println("Hello " + name);
        ms.sendGreeting("Steve");
    }//end main
}//end LambdaDemo
```

- NOTE: how abstract method in interface determines the structure of the lambda function — the parameter and parameter type, the method name and the lack of a return statement
- These functions could be useful (especially when attaching events to GUI components)
- Only certain interface structures can be used
- Suppose you have a method the only displays certain members of a collection, depending on a criteria that is specified outside the method and provided as an argument
- You don't want to hard code the criteria in the display method

- FIRST technique: Supply an object implementing an interface that has a "filter" method that returns a true or false.
- SECOND technique: Use an anonymous class instead
- THIRD technique: Use a Lambda function

But how does it work?

The structure of the method implemented by the lambda function is specified by the interface type used in the displaySome method. The signature is:

```
boolean check(Person)
```

- The compiler knows from this that the type to the left of the -> must be a Person and the expression to the right of the -> must return a boolean

- Filter is an example of a **functional interface**
- These interfaces can only contain a single abstract method
- Lambdas can only be created using Functional interfaces
- You can use the @FunctionalInterface annotation to make sure your interface is OK

- You can have multiple lines of code in a lambda, but don't make them too long
- You can use as many lambdas as you wish when invoking a function, as long as they each match up to some functional interface
- In GUI programming the most common event handler interface, EventHandler<ActionEvent> is a functional interface so lambdas can be used to attach event code to handlers

Pre-Defined Functional Interfaces

- Turns out java.util function package has many pre-defined generic functional interfaces
- The one that matches our check function signature is called Predicate<T>. It has the abstract method signature: boolean test(T)

Method References

- ArrayList.sort() accepts a Comparator<T> object that can specify the desired algorithm for comparison of objects of type T
- Turns out Comparator is a FUNCTIONAL INTERFACE, so you can build a lambda function for a comparator

- But suppose our Person class has a method that matches the Comparator interface:

```
public static int comapreByAge(Person p1, Person p2) {
        return p1.age - p2.age;
}
```

- in this case, you can supply a METHOD REFERENCE instead of building a lambda function

```
db.sort(Person: : compareByAge);
```

- The method reference must match the functional interface's specification
- You can use non-static methods or methods from an instance. If you wish to supply a constructor, use:

```
ClassName: : new
```

- and you can use existing API methods

GUI Programming

Layout Managers

- Each pane is associated with a layout manager — an algorithm that determines which controls are placed within the pane and often their size
- Different panes have different managers
- Provides flexibility during design and during runtime when the size of window is changed or window is viewed on a different OS than the one which it was designed

BorderPane

- Lays out components in top, let, right, bottom, and centre positions

Example: to add a Button to the Top position:

```
BorderPane bPane = new BorderPane();
Button myButton = new Button("Click me");
bPane.setTop(myButton);
```

- A component adopts its preferred size if it is smaller than the available region
- Center swells to occupy neighbouring positions if they are empty

- You can add margins round a component by using an Insets object with the static .setMargin() method
- You can also specify alignment within a position using a value from the Pos enum and the static .setAlignment() method
- This layout is good for blocking out an entire window into sub-regions and then you use another layout in the regions of interest

Anchor Pane

- Components are anchored at a specified distance from the edge of the Pane using the static methods:
  - setBottomAnchor(Node child, Double value)

(along with Left, Right, and Top anchor)

- "value" is in pixels as a distance from the border of the pane
- Note: the effect of window re-sizing
- Note: how two anchors can combine to change the size of the control away from its calculated/ preferred size
- This is another relatively crude layout manager that might be best used to lay panes out in a window

Pane "Children"

- Nodes added to a pane are also called "Children"
- Add nodes to a pane by invoking the .add() or .addAll() methods on the children collection owned by the pane. The .addAll() method can also accept a Collection<? extends Node> object
- The .getChildren() method of the pane object "exposes" its children collection

Flow Pane

- Piles children into the pane in the order in which they are added form left to right (by default)
- The children wrap around if the available space is full
- Note: the use of an ArrayList<Button> to hold buttons

Grid Pane
- Possibly the hardest Pane to use but it offers the most control
- Grid based — you need to choose which position in the grid you will use for your component
- A column is sized to the widest component in that column
- Similarly the height of a row is determined by the tallest component in the row
- You can add internal gaps between rows and columns using .setHgap() and .setVgap()
- Set gaps on the outside of the pane using .setPadding() and supply an Insets object
- The pane has two .add() methods you can use, one at a time to add components:
  .add(node, column#, row#)
  .add(node, column#, row#, columnSpan, rowSpan)
- During development invoke .setGridLinesVisible(true) to see the lines in the grid

HBox and VBox Panes
- Pretty straight forward, less annoying than FlowPane (no wrapping)
- These would either be used for very simple designs or within a region of another layout

Tile Pane
- Another grid based layout, but simpler than GridPane
- Controls are laid down in the order in which they are added
- you can specify a desired number of rows and/or columns but the actual layout may change if needed
- Column widths and row heights are fixed to the largest control in that column or row (like GridPane)

Pane
- The root class for all the other pane classes
- You can use this class, but it does not have a layout manager
- All positioning is manual. You need to manually size (height, width) the component and specify the (x, y) positions (ouch!)

Getting a Button to do something
- Use the .setOnAction() mutators to add an event handler to the button
- The handler for a mouse click event must implement EventHandler<ActionEvent>

Four ways of doing this:
- Another separate class (we skipped this)
- A private inner class
- An anonymous class
- A Lambda function

event Parameter
- Is of a type that extends Event — the base class for all events in JavaFX

Change Events
- You can also respond to a change in the contents or the state of a Node (think TextField or ChoiceBox, or Slider etc)
- To do this on a particular node:
  node.valueProperty().addListener(…)
- Then create a lambda using the ChangeListener<> interface