Date	Lecture Topics	Code	On Your Own

This document is updated frequently, so remember to refresh your browser.

Section numbers refer to the current version of the textbook (Java Concepts 8th edition or Big Java 6th edition). If you are using an older version of the book, see the section number mapping document.

Chapters 13, 14, and 15 of Java Concepts, 7th edition, can be found here.

How to import the code examples below into an Eclipse project.

Week 15			
April 29	• "Unwinding" the call stack to find the first matching `catch` block • Cleanup: `try/catch/finally` and `try/finally`	ExceptionExample.java	• There will be a **group quiz** on exceptions. Read Section 11.4.
May 1	• Fun with sockets (realistic examples using try/catch and try/catch/finally)	SimpleServer.java SimpleClient.java SimpleHttpServer.java SimpleHttpClient.java	•
May 3	• Review		•

Week 14			

April 22	• Subtypes are created by implementing an interface or extending a class
	  ◦ The purpose of an *interface* is to reduce coupling between components by eliminating explicit dependence on a particular class
	  ◦ The purpose of *inheritance* is to re-use existing code while allowing for specialization
	• Polymorphism works the same way for both: *the code that actually executes depends on the runtime type*
	• Case study: redesigning the `SVFunction` hierarchy so that polymorphism can be used when accessing the function description and parameters.
		graphing_calc_better.zip
		See the `ugly.GraphingCalculator` for example using explicit type-checking with `instanceof` and downcasts. See the `better` package for a redesigned solution using an interface to describe the function parameters.
		Here are some notes on our redesign of GraphingCalculator.
		Here is the class diagram for the SVFunction hierarchy.
		Animals.java
		And here is the class diagram for the Animals hierarchy.	

	• The `instanceof` keyword: use sparingly		

April 24	• Don't drink too much Kool-Aid
	  ◦ Inheritance isn't always the right solution
	• Understanding inheritance vs. composition
	• *is-a* vs *has-a*
	  ◦ A Car *is a* Vehicle
	  ◦ A Car *has a* Wheel
		NoDupListWithInheritance.java
		NoDupListWithComposition.java	

April 26	• Exceptions
	  ◦ *Reporting* errors: the `throw` statement
	  ◦ *Handling* errors: `try/catch` blocks
	  ◦ *Declaring* errors: the `throws` clause (required for "checked" exception types only)
	• The Exception class hierarchy; "checked" vs "unchecked" exceptions
	  ◦ The "unchecked" exceptions are subtypes of RuntimeException
	  ◦ "checked" exceptions must be caught or declared with `throws`
		ExceptionTest.java
		ExceptionHandlingExample.java
			• Read Section 11.4

Week 13			

April 15	• Inheritance
	• A subclass "inherits" the attributes and methods of its parent class
	  ◦ It inherits the instance variables, even though they are not *accessible* in the subclass because they are private
	• A subclass must call the superclass constructor to initialize the superclass instance variables
	  ◦ The call to `super()` **must** be the first
		ZooInheritanceTest.java
		Contact.java
		PhoneContact.java
			• There will be a **group quiz** on interfaces. Read the beginning of Chapter 10 (at least through Section 10.1).
			• Read 10.3 and 10.4
			
			Note: chapters 9 and 10 are reversed in older editions of the textbook.

	statement in the subclass constructor ◦ If the superclass has a no-arg constructor, the compiler can insert the call to `super()` automatically • A subclass can *override* methods from the superclass or add new methods of its own		
April 17	• Inheritance, con't • **Never** re-declare instance variables in a subclass ◦ Use the superclass constructor to initialize them ◦ Use the public API methods to access them • Dynamic binding, con't: the code that executes for a method call `v.m()` depends on the *runtime* type T of the variable `v` ◦ If there is no implementation of `m()` in class T, then the VM looks in T's superclass, and then that class's superclass, and so on up the chain... ◦ ...to `java.lang.Object` • Every Java object has one unique "parent" (direct superclass), except `Object`, which is the root of every class hierarchy ◦ The `super` keyword can be used to call the superclass methods • Every Java object is a subtype of `Object` and inherits methods such as `toString`, `equals`, and `getClass`.	Contact.java PhoneContact.java TextContact.java EmailContact.java EmergencyDatabase.java EmergencyDatabaseTest.java Here is the class diagram for the Contact hierarchy. (Diagram generated using the ObjectAid plugin for Eclipse.)	
April 19	• An `abstract` method has no body ◦ interface methods are always `public` and `abstract` by default	• In case your edition of the text does not have it, the "Special Topic 9.1" section on abstract classes can be found here	• There will be a **group quiz** on the basics of inheritance (Sections 9.1-9.3) • Read 9.5

	• If a class contains one or more `abstract` methods, then it must be declared `abstract` as well
	• Abstract classes cannot be instantiated
	• An abstract class is a partial implementation that needs to be extended
	• Use an abstract class for code that is common to two or more subclasses when it doesn't make sense for those classes to extend each other
	  ◦ Neither `Cat extends Dog` nor `Dog extends Cat` makes any sense
	  ◦ Instead, `Cat` and `Dog` can extend an abstract class `Pet` with the common code
	• The `protected` keyword
	  ◦ Avoid `protected` variables
	  ◦ Use a `protected` method when your intention is that subclasses should override it
	  ◦ An abstract class usually has protected constructors, since only subclasses can call them
	• *is-a* vs *has-a*
	  ◦ A Car *is a* Vehicle (suggests *inheritance*)
	  ◦ A Car *has a* Wheel (suggests *composition*)
			Note: chapters 9 and 10 are reversed in older editions of the textbook.

Week 12			
April 8	• Review		
April 10	• Interfaces are used to "decouple" components • The interface describes a *capability* that an object must have for a component to use it • Examples:	FirstSwingExample.java Contact.java ContactSorter.java graphing_calculator.zip (This is a complete Eclipse project you can import; alternatively you can create a new project and drag the packages	*Note: chapters 9 and 10 are reversed in older editions of the textbook.*

	◦ The Zoo can contain any kind of animal, as long as it implements the `Creature` interface ◦ The Swing GUI framework can invoke any event handler in our code, provided that it implements the `ActionListener` interface ◦ The Java `Arrays.sort` method can sort any type of object that implements the `Comparable` interface ◦ The `GraphingCalculator` can plot any function, provided that it implements the `SVFunction` interface • Declared type vs runtime type ◦ *Polymorphism* says that a variable `v` declared with type T can refer to an object of any subtype S of T ◦ T is called the *declared type* the variable (never changes) ◦ The *runtime type* of the variable is the type of the actual object it refers to when the program executes (can change from instant to instant) ◦ **The code that executes when you call a method is always determined by the runtime type**	into the src folder. There are main() methods to run in `bad.GCMain` and in `good.GCMain`.)	
April 12	• No class :)		

Week 11			
April 1	• More examples with	FileListerExamples.java	• The permutation

	recursion - problems that are easy with recursion, but difficult with loops. ◦ Permutations ◦ Searching a file hierarchy • Recursion with arrays • The divide-and-conquer strategy ◦ Split the problem into two parts ◦ Solve each part (usually using recursion) ◦ Combine the parts into a solution	Permutations.java ArraySum.java Here is a version of ArraySum that prints out what it's doing at each recursive call... ArraySumVerbose.java	example is also done in Chapter 13
April 3	• The divide-and-conquer strategy, con't • Binary search • Merge sort • Profiling merge sort vs selection sort	BinarySearch.java MergeSort.java Profiling selection sort vs merge sort: Test.java SelectionSort.java Alternate implementations of merge sort that reduces memory use: MergeSortAlt2.java	• Read 14.1 and 14.4 • Binary search is discussed in Section 14.6
April 5	• Tail recursion ◦ A method is *tail recursive* if the recursive call is the last thing that happens ◦ It's easy to convert a tail recursive method into a nonrecursive implementation (see BinarySearch) • Why recursion is good for merge sort but bad for Fibonacci numbers • Introducing Java interfaces • *Polymorphism*: A variable `v` declared with type T can refer to an object that is a *subtype* of T ◦ In `ZooTest.java`, a variable of type `Creature` can refer to any class implementing the `Creature` interface	FibonacciTest.java ZooTest.java	• Read Section 10.1 on interfaces (Note that in older editions of the textbook, interfaces are covered in Chapter 9.)

Week 10			
March	• Two-dimensional arrays	ArrayExamples2D.java	

25	◦ The first index is always the row! ◦ In Java, a 2D array is really an array of arrays ◦ Each element is one row (a 1D array) ◦ The number of rows is `arr.length` ◦ The number of columns is `arr[0].length` • Making a copy of an array ◦ *shallow copy* - copy array values only ◦ *deep copy* - make copies of the objects they refer too as well ◦ (an assignment never makes a copy of anything in Java) • Methods such as `Arrays.copyOf` always make a shallow copy • To make a deep copy, you generally have to use a constructor to make a duplicate of each object in the array ◦ Be careful when using `clone()` since not all objects are implemented to support cloning • Don't use `Arrays.copyOf` for 2D arrays	ArrayDeepCopy.java ArrayCopy2DArray.java	
March 27	• *Recursion* is when a method calls itself. ◦ Every method call has its own local variables ◦ Local variables live in an *activation frame* on the call stack ◦ The frame includes a "return address", where to resume execution when the method returns • Thinking recursively ◦ Reduce the problem to a smaller instance ◦ Identify a base case ◦ Believe that it will work	Step through the `countUp` method in the debugger (or pythontutor) to visualize the frames in the call stack. RecursionExamples.java PalindromeChecker.java InfiniteRecursion.java	• Read Chapter 13. Note that for *Java Concepts*, Chapter 13 is found online. See the link at the top of this document.

	• Our initial examples of recursion are simple problems that we could easily solve with loops... • ...however, it turns out to be a powerful problem-solving technique for many problems that are very difficult to solve with loops • Infinite recursion and `StackOverflowError` • Using extra parameters in `PalindromeChecker` to keep track of the start and end of the substring		
March 29	• Some problems that are very hard to write with loops are very simple using recursion	TowersOfHanoi.java JimGoesUpStairs.java	• There will be a **group quiz** on recursion (Chapter 13).

Week 9			
March 11	• The idea of a nested loop • Partially filled arrays • Example: the class `IntList` ◦ This is how `ArrayList` is implemented behind the scenes • Swapping elements • The selection sort algorithm	NestedLoopExample.java IntList.java IntListTest.java SelectionSorter.java	• Read 6.8 • Selection sort is discussed in Section 14.1, which you can find online here
March 13	• The class `ArrayList` • Generic type parameters in Java classes • Using an ArrayList for reading input • Comparing arrays and ArrayLists • The `toArray` method • Wrapper classes for primitive types	ArrayListExample.java FindMedianWithArrayList.java FairDieChecker.java FairDieCheckerWithArrayList.java	• There will be a **group quiz** on ArrayList. Read Section 7.7 of the text.
March 15	• Avoid parallel arrays! Define an object type to keep related values together ◦ See Note.java in the JohnnyOneNote example • An ArrayList is especially useful for reading data from a file	ArrayRemoveOddNumbers.java JohnnyOneNotePrototype.java **Files for the improved JohnnyOneNote:** JohnnyOneNote.java Note.java SongReader.java PlayerMain.java mary.txt	• See 7.3 and 7.7

	• More examples of array modification
	  ◦ Returning a new array vs modifying a given array
	  ◦ Doing repeated deletions while iterating is tricky
	  ◦ In some cases (e.g. moving even numbers to front) the modification can be done "in place" (without creating a temporary array)
	  ◦ Shift even numbers to front, odd numbers to end: to avoid clobbering all the odd numbers, can use an auxiliary ArrayList to store them		

Week 8			

March 4	• Using a while-loop to parse input
	• Reading text files:
	  ◦ Constructing a Scanner with a File object opens the file
	  ◦ File path is described by a `File` object, relative to the project directory
	  ◦ Method that opens file needs the declaration `throws FileNotFoundException`
	  ◦ Remember to close the Scanner
	• An exception like `FileNotFoundException` that requires the `throws` declaration is called a "checked" exception
	  ◦ the declaration says, in effect, "this method *might* cause this type of exception to occur"
	• Text files are usually organized into lines ended by one of the platform-
		LoopToParseInput.java
		LoopToReadFile.java
		FileTest.java
		ReadFileLineByLine.java
		(For the examples above you'll need a text file in the **project directory** containing some data to read, e.g. nums.txt and scores.txt)
			• Read Sections 11.1 and 11.2

	dependent *line termination characters* • To read a file line by line, use the scanner methods `hasNextLine()` and `nextLine`		

March 6	• Arrays!
	  ◦ An array is a sequence of elements that can be accessed by index (like characters in a string)
	  ◦ Can be of any Java type
	  ◦ Fixed in length (but length can be computed during runtime)
	  ◦ Mutable
	  ◦ Arrays are objects and are created with the `new` keyword
	  ◦ A newly created array has default values for all elements (zeros or nulls)
	  ◦ There are no useful built-in methods, use the `.length` attribute to get the length
	• The `Arrays` class has many static methods for working with arrays, such as `toString` and `sort`
	• Reading input into an array: need to first determine the length, then create the array
		ArrayExamples.java
		FindMedian.java
			• There will be a **group quiz** on arrays, covering Sections 7.1 and 7.3.
			• Read 7.1 - 7.3

March 8	• Many of the kinds of loops we write with strings are the same for arrays • The "foreach" loop (or "enhanced for loop") • Some standard array algorithms: sum, average, maximum, minimum, find index of an element • Methods that modify an array • Deleting an element (and shifting left) • Inserting an element (and shifting right)	ArrayLoopExamples.java MoreArrayExamples.java	• Read 7.4, 7.5

Week 7			
February 25	• Review		
February 27	• Recap of while-loops vs. for-loops • Hand-tracing a loop • The idea of a "search" problem ◦ Once you find something, don't un-find it! • Using a `return` statement in a loop • Using the `break` keyword • (The `continue` keyword: *just say No*)	SearchProblems.java	• Read 6.5 and 6.7 • Section 6.2 is an example of hand-tracing, using a table to keep track of variable values at the end of each iteration • "Special topic 6.3" (in Section 6.5) discusses use of a boolean flag to control a loop • "Special topic 6.4" (in Section 6.5) discusses `break` and `continue`
March 1	• No class		

Week 6			
February 18	• The boolean operators `&&` (*and*), `||` (*or*), and `!` (*not*) • De Morgan's Laws	GradeCalculatorWithInputCheck.java LeapYearCalculator.java	• See Section 5.7
February 20	• Loops! • The `while` statement • Identifying initialization, condition, and body of a loop	WhileLoopExamples.java GuessingGameWithLoop.java GuessingGameWithLoopMain.java	• Read 6.1 and 6.2.

February 22	• The `for` statement
	• While-loops vs. for-loops
	  ◦ Use `for`-loops when counting with an index or iterating over a fixed sequence of items
	  ◦ Use `while`-loops when the number of iterations is unknown or difficult to compute
	• Scope of variables declared within a block
		ForLoopExamples.java	• Read Chapter 6. There will be a **group quiz** on while-loops as in the examples in 6.1 and 6.2.

Week 5			
February 11	• Conditional statements in Java • Relational operators `<`, `<=` and so on ◦ You can test equality of `int` values with `==` or `!=`, but these are not useful for floating-point numbers or objects! • Problem-solving strategy #1: *do a concrete example first!* • Describing program logic with pseudocode	DoorWithLock.java Order.java OrderTest.java	• Read Chapter 5 • See the "How To 4.1" in Section 4.3

February 13	• Using `==` for objects only tests for *identity*, not "sameness".
	• When to use `==` (or `!=`)
	  ◦ For integer and char types
	  ◦ Checking whether an object reference is `null`
	  ◦ For `enum` types (we'll see them later on)
	  ◦ Use `Math.abs(d1 - d2) < ERR` to test whether two `double` values are the same
	  ◦ Use `.equals()` for strings
		FloatingPointEqualsTest.java
		OrderUI.java
		StringEqualsTest.java
			• Read Chapter 5. There will be a **group quiz** on basic conditional statements as in 5.1.

February 15	• Nested conditional statements
	• Multiway branches with the "else if" idiom
	• Sequential vs nested `if` statements: do we want to "fall through", or are the alternatives mutually exclusive?
	• Using sequential `if` statements by mistake
	  ◦ See `getLetterGradeWrong` in `GradeCalculator`
	• Using "helper" methods to simplify a program
	• The `static` keyword
	  ◦ A method may be declared `static` if it does not refer to any instance variables
		MaxOfThree.java
		GradeCalculator.java
		GuessingGame.java
		GuessingGameMain.java
			• Read 5.3 - 5.5

	▪ Example: utility methods such as `Math.max` ◦ We usually declare *constants* to be `static final` ◦ Using `static` for *variables* is usually a mistake		

Week 4			
February 4	• Incremental development ◦ You don't have to solve everything all at once! ◦ Implement one feature at a time, and test as you go ◦ *Test early, test often!*	GasTank.java GasTankTest.java	
February 6	• Incremental development, con't • Start with simple, concrete examples that you can do on paper (or on your fingers) ◦ The steps of a simple example show you how to write the code ◦ You can use your examples for test cases too • An object can have instance variables that are also objects ◦ ...remember to initialize them! • Introducing `Scanner` for parsing input	Car.java CarTest.java ScannerTest1.java ScannerTest2.java ScannerTest3.java	• Read Sections 3.5 and 3.6
February 8	• Guidelines for instance variables: ◦ Avoid redundant instance variables (Common Error 3.4 in Section 3.6) ◦ Don't declare an instance variable for a value that can be declared locally within a method ◦ Make sure every instance variable is	BadSimpleRectangle.java BadSimpleRectangleTest.java BadCar1.java BadCar1Test.java BadCar2.java BadCar2Test.java BadCar3.java BadCar3Test.java	• There will be a **group quiz** on basic use of Scanner. See Sections 4.3.1 and 4.5 3, and the Scanner examples posted last time.

	initialized in the constructor ◦ All instance variables should have valid values at the end of each method ◦ Accessor methods should never modify instance variables ◦ Avoid shadowing (don't name a local variable the same as an instance variable) (Common Error 3.3 in Section 3.6) • Default values for instance variables ◦ Numeric variables default to zero ◦ Boolean variables default to false ◦ Object variables default to `null` • Interpreting a `NullPointerException` (Common Error 3.5 in Section 3.6)		

Week 3			
January 28	• *Client view:* the API (public) • *Developer view:* instance variables, method implementations (private) • *Encapsulation*: data and implementation details are hidden from clients ◦ Clients only depend on the public API • Implementing our own class! ◦ *Instance variables* represent the object's state ◦ *Constructors* initialize the instance variables ◦ *Methods* provide the operations we want • Mutator vs accessor methods • Documenting a class using Javadoc comments	SimpleRectangle.java SimpleRectangleTest.java	• Read Sections 3.1, 3.2, and 3.3
January	• No class, university		

30	closed. Brrrr!		

February 1	• Recap: a typical class definition has
	  ◦ *Instance variables* - the object's state
	  ◦ *Constructors* - initialize the instance variables
	  ◦ *Methods* - the operations we want
	• Methods are usually either
	  ◦ *Accessors* - return information about the object's state (without modifying anything)
	  ◦ *Mutators* - modify the object's state (usually return void)
	• Typical steps for defining a class
	  1. Decide the operations that are needed
	  2. Define the public API as Java methods and constructors
	  3. Write stubs for the methods and constructors
	  ▪ Start writing test cases
	  4. Define instance variables and initialize in constructors
	  ▪ Look at the accessor methods for inspiration
	  5. Write the method bodies
	  6. Test
	• Always start with simple test cases or usage examples before writing too much code!
		Door.java
		DoorTest.java
			• There will be a **group quiz** on Friday covering 3.1 - 3.3. You will be expected to write a class definition similar to the examples above or in the textbook.
			• See the "How To" at the end of section 3.3
			• Read 3.4 and 3.5

Week 2			
January 21	• No class		
January 23	• The primitive types `int`, `double`, `char`, and `boolean`	VariableTest.java StringTest.java	• Read Sections 2.1, 2.2 and 2.3

	• The `String` type
	• Variable declarations
	• The assignment operator `=` (*"gets the value of..."*)
	  ◦ For the `int` type, you can use a double-equals operator `==` to mean *"is equal to..."*
	• Anatomy of a method call
	• Number and types of arguments must match the method's definition
			• Check out the Java API documentation.
			  ◦ For example, look for the `String` class in the package `java.lang`. It's pretty overwhelming right now, but soon, the API docs will be your new Best Friend.

January 25	• Using a constructor with the `new` keyword to create an object • Visualizing object references	RectangleTest.java	• There will be a **group quiz** covering 2.1 - 2.3. You'll be asked to write a class with a `main` method that calls some methods on a variable of type String (such as the examples in the text). • Read 2.4 and 2.5

Week 1			
January 14	• Introduction • Who should take Com S 227? • Syllabus overview *Powerpoint slides from first day*		• Log in to Canvas! • Read the syllabus! • Sign up for Piazza!

January 16	• Writing an algorithm in English or "pseudocode".
	  ◦ In this course we are also concerned with how to organize *data* into "objects"
	  ◦ In a nutshell, an *object* is a container for data together with the operations that act on it
	• Pseudocode and code examples (pdf)
	• (For more examples of a simple algorithm implemented in many
		Greeter.java	• Read Chapter 1

	different programming languages, see 99 Bottles of Beer). • Writing our first Java program *Powerpoint slides from second day*		

January 18	• Role of the compiler and the Java Virtual Machine (Java runtime)
	• Basic Java keywords and syntax
	• A *class* is a definition for a type of object
	  ◦ In our "Hello, World" example, the class itself serves no purpose except as a container for the `main` method
	  ◦ This is not typical of Java classes!
	• Values and types
	• Arithmetic operators
	• Integer division and the *mod* operator `%`
		ExpressionTest.java	