**Remember:**

* You cannot access a non-static variable or method from a static method. So you can’t access anything that’s not static from your main method (which is static).
  + Instead, you have to create an object of that class to call the static method on.
* If you try to override a non-static method with a static method: COMPILER ERROR
* If you try to override a static method with a non-static method: COMPILER ERROR
* If you try to override a static method with a static method: No errors unless you do @Override, in which case COMPILER ERROR

**Questions:**

* If an overloading method has the same name and parameter input (signature?) but different return type, is this an error? What if the overloading method is in the child class?
  + DOES NOT COMPILE
* Question 11.19 (last paragraph before it as well)
  + ALL OF THEM WORK EXCEPT FOR INT BECAUSE IT DOESN’T HAVE A CLASS.
* 11.20
* Problem 11.23; so when you call new A(); it actually creates an object and initializes all the values? What happens to all that stuff after since there’s no reference for it?
  + SO ANY METHOD THAT I PUT IN A BUT IS OVERWRITTEN BY B WOULD BE RUN BY THE B CLASS WHEN I CALL NEW B();?
  + IT CALLS IT FROM THE CONTEXT OF B
* 22.1, what’s the difference in reason?
* I don’t understand example 7 in Chapter 22.3
* 23.3
* Verify 11.25 & 11.26
* 11.29, you haven’t actually cast the this object. Isn’t it still of class object?
* The proper format for an ArrayList. Also the method returns are incorrect.

Answered myself:

* So when you explicitly cast something to a subclass, does it call the constructor of that subclass too?

**Notes:**

* String[] args has a default delimiter of space or “ and ‘. So it will separate anything between two “s or ‘s in the array.
* Every letter in a java keyword is lowercase
* Remember in a

Introduction to Java Programming – Notes

**Chapters Covered**

* **Chapter 9:** Objects and Classes
* …?

Most important chapters

* **Chapter 11:** Inheritance and Polymorphism
* **Chapter 12:** Exception handling and Text I/O
* **Chapter 13:** Abstract Classes and Interfaces

# Chapter 9: Objects and Classes

## 9.4 – Constructing Objects Using Constructors

* Constructors are a special kind of method that”
  + Must have the same name as the class itself
  + Do not have a return type, not even void
  + Are invoked using the *new* operator when an object is created, constructors initialize the object
* They can be overloaded (same name but different signatures), so objects with different initial values can be created
* The constructor is called when an object of the calass is constructed with *new*:
* Usually, a no-argument constructor is created
* A class can be defined without any constructor, in which case a public no-arg constructor is implicitly defined
* **I think that if you have multiple constructors then the first line within them should be a chained call to another constructor or it will error**
  + **This is true even if you try to place a super() call as the first line (instead of a this(…) call)**
  + **But the problem is that super also needs to be the first line**
  + **Which is all pretty weird because if you don’t write super() or super(parameters) in your child constructor, then it automatically insterts super() as the first line, which works fine.. but you can’t manually insert super()…?**
* **Note from a later chapter:** If you define a constructor other than the no-argument constructor, then the no-arg constructor is NOT implicitly defined for you, it just won’t exist. The point is that there must be at least one constructor, and if you define one then it’s all good so no need to implicitly do so.
* **Note 2:** This is pretty amazing but, say you have class A and its constructor calls method1, which is also defined inside A. Then you have class B that extends A and overrides method1 but you don’t define any constructors (or you just define a no-arg constructor that just calls super()). Then when you instantiate an object of type B, it will super to A’s constructor, which will call method1. But which one? Well if the dynamic type of the object is B, it will actually call method1 from B!!

## 9.9 – Data Field Encapsulation:

* Making data fields private protects the data
* Making a data field private is known as *data field encapsulation*

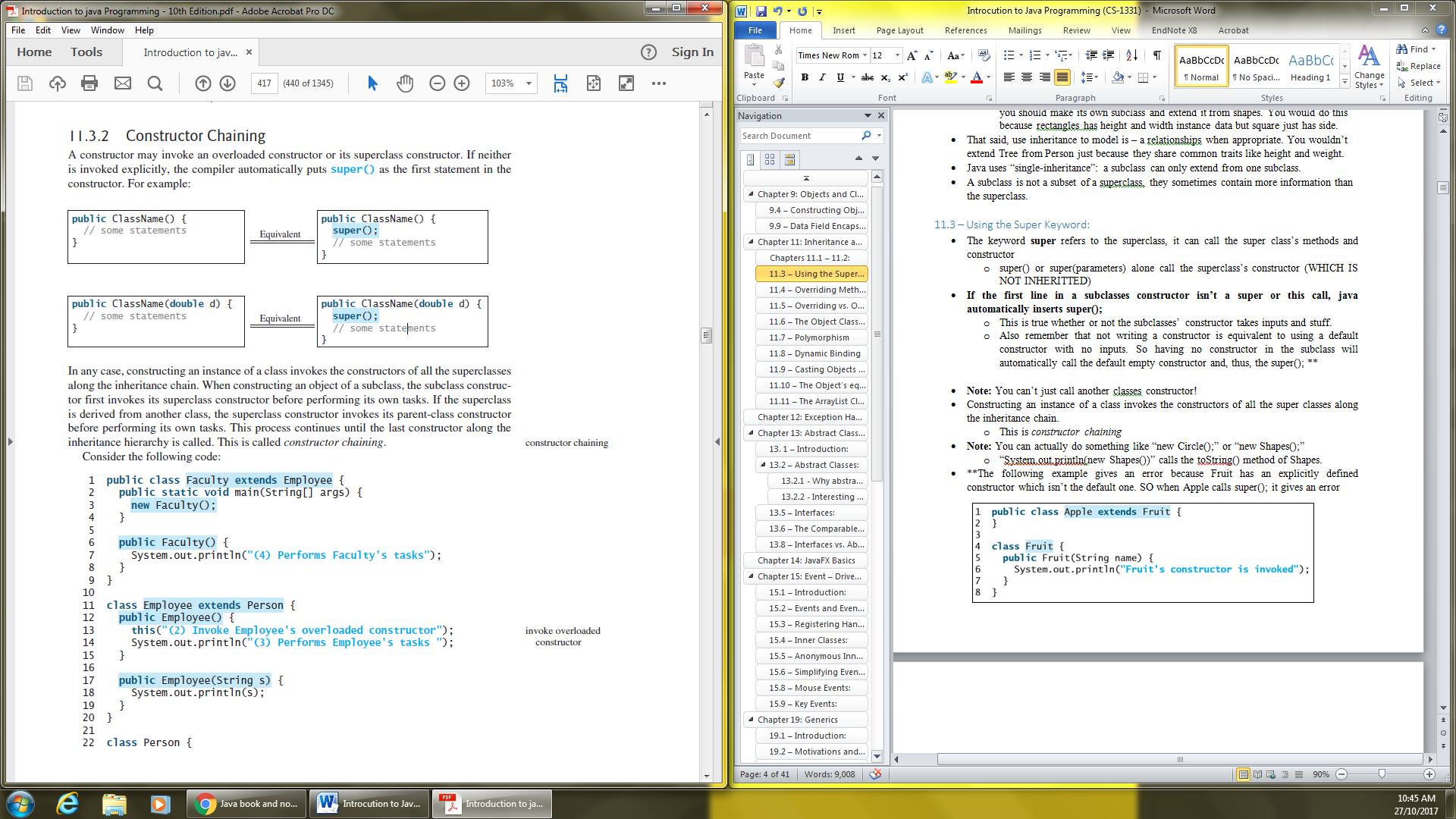
# Chapter 11: Inheritance and Polymorphism

## Chapters 11.1 – 11.2:

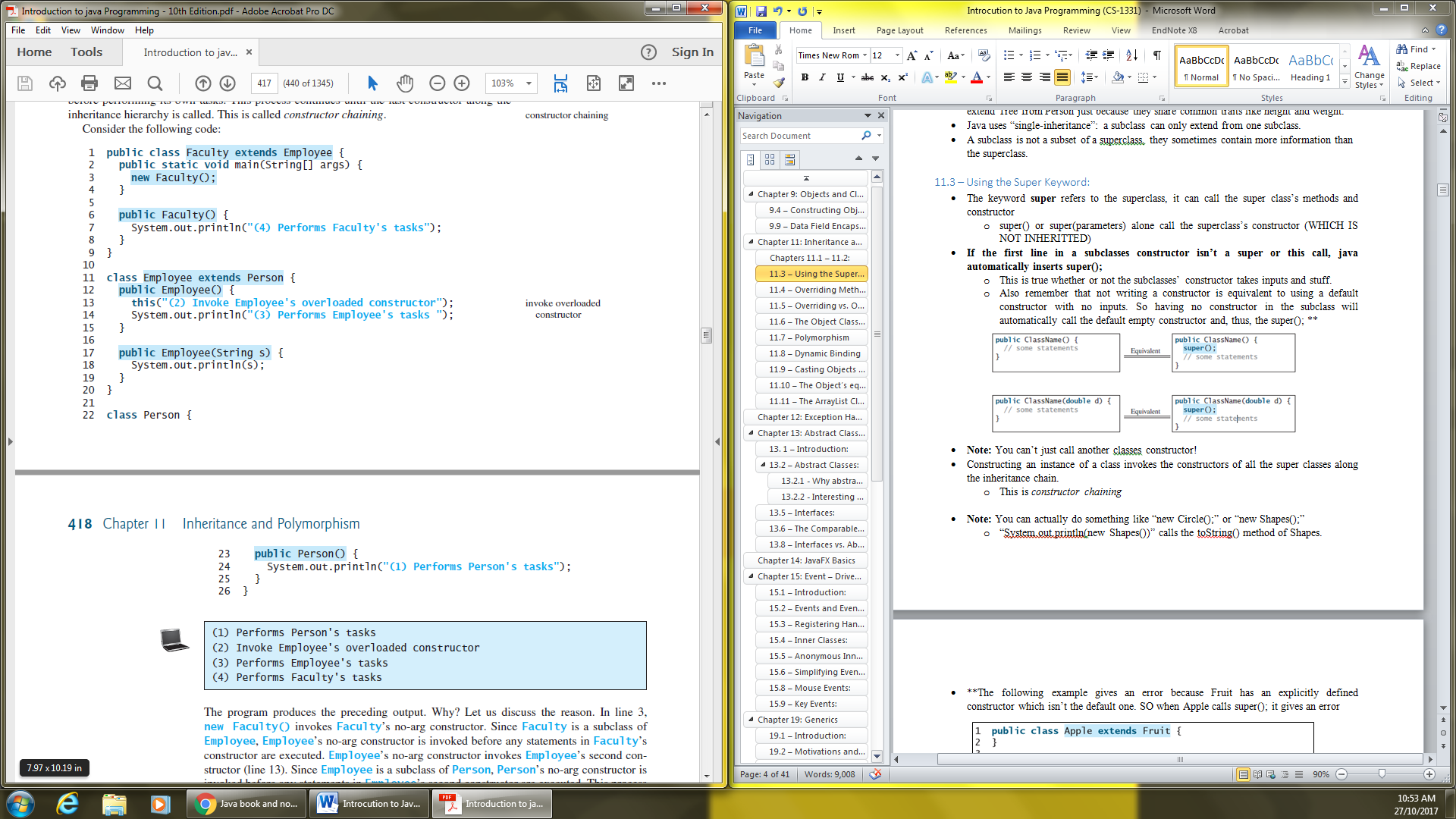
* Inheritance allows you to define a general class and then extend it to more specialized subclasses
* …
* Not all is-a relationships should be modeled with inheritance.
  + For example, a square is a rectangle, but you shouldn’t extend it from rectangle, you should make its own subclass and extend it from shapes. You would do this because rectangles has height and width instance data but square just has side.
* That said, use inheritance to model is – a relationships when appropriate. You wouldn’t extend Tree from Person just because they share common traits like height and weight.
* Java uses “single-inheritance”: a subclass can only extend from one subclass.
* A subclass is not a subset of a superclass, they sometimes contain more information than the superclass.

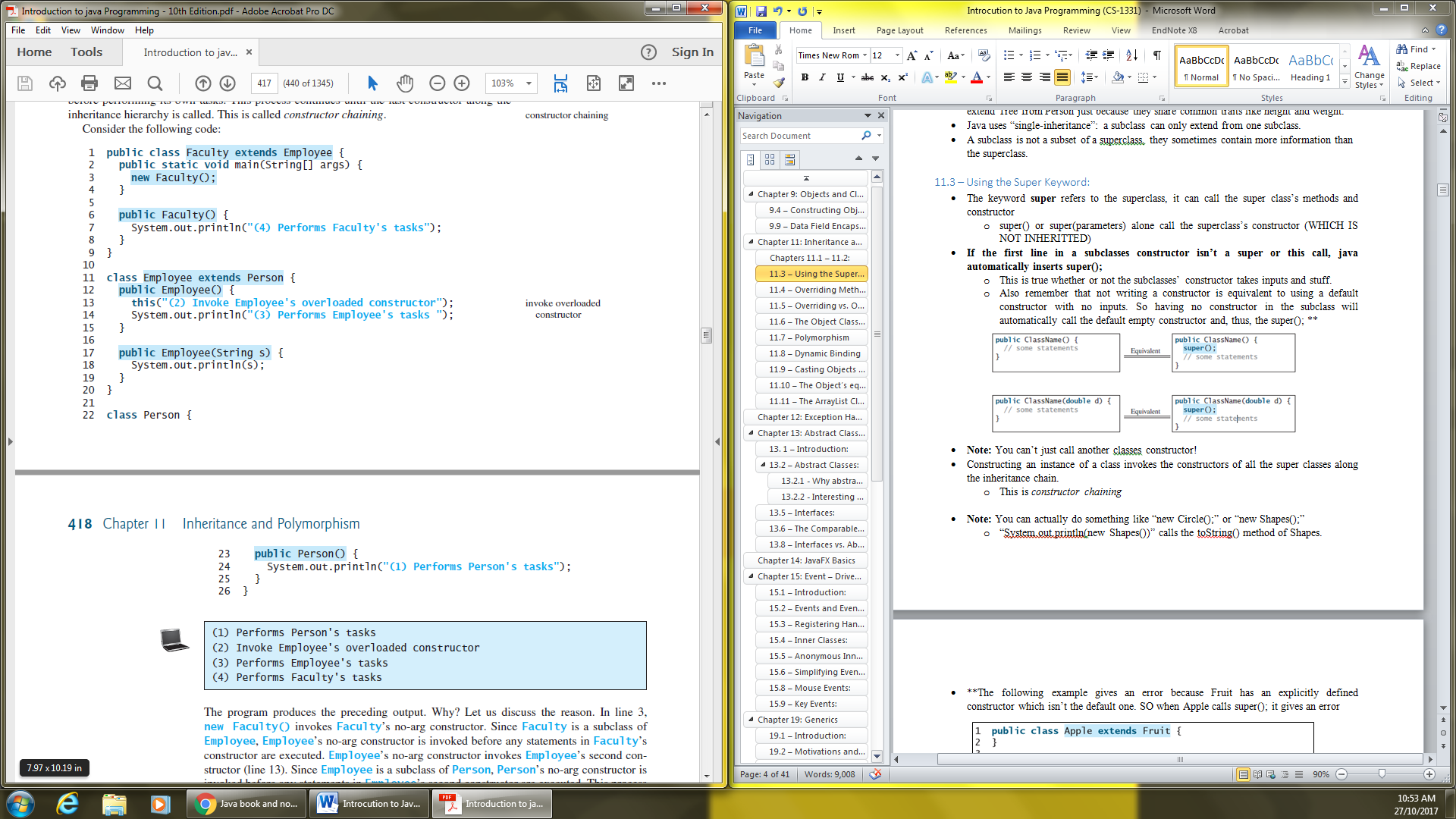
## 11.3 – Using the Super Keyword:

* The keyword **super** refers to the superclass, it can call the super class’s methods and constructor
  + super() or super(parameters) alone call the superclass’s constructor (WHICH IS NOT INHERITTED)
* **If the first line in a subclasses constructor isn’t a super or this call, java automatically inserts super();**
  + So basically, for all constructors in a child class, the first line should be a super() or super(parameters
  + This is true whether or not the subclasses’ constructor takes inputs and stuff.
  + Also remember that not writing a constructor is equivalent to using a default constructor with no inputs. So having no constructor in the subclass will automatically call the default empty constructor and, thus, the super(); \*\*

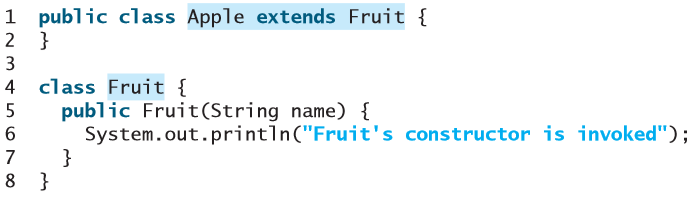


* **Note:** You can’t just call another classes constructor!
* Constructing an instance of a class invokes the constructors of all the super classes along the inheritance chain.
  + This is *constructor chaining*





* **Note:** You can actually do something like “new Circle();” or “new Shapes();”
  + “System.out.println(new Shapes())” calls the toString() method of Shapes.
* \*\*The following example gives an error because Fruit has an explicitly defined constructor which isn’t the default one. SO when Apple calls super(); it gives an error



**ERROR:** “Constructor Fruits in class Fruits cannot be applied to given types”

* You can also call the super classes methods: super.method();

## 11.4 – Overriding Methods:

* You override a method in the subclass by giving it the same name and parameter signature.
* You can’t do “super.super.method();”: syntax error
  + Instead, make a method in the super class that calls it.
* “A private method cannot be overridden”; what this is saying is that, if a method is private and a subclass defines a similar method, then the two methods will be completely unrelated. **You don’t get a compiler or runtime error.** It’s just that it’s not technically “overridden” because the subclass can’t see the private method in the superclass.
  + **Note:** Remember also that you can’t access a private method by just calling it externally on an object of that class: object.privateMethod(); will return: “ERROR: privateMethod() has private access in object” if the object is of the class where privateMethod() is in. You instead have to make a public method in object that calls privateMethod().
  + **Note:** Overriding a private class(method I think actually) will cause an error if, in an external class, you create an object as such: “ParentClass object = new ChildClass();” and then call the private method on it **(i.e. the error arises when you call “object.privateParentMethod();”)** because the method is private in the parent class.
  + **Note:** It’s not overriding either, but, in your child class, you can rewrite the method (not overriding) with the same name, parameters, etc. and make it public so that, when you call that method on an object of that class (“ChildClass object = new ChildClass();” and “object.publicChildMethod();” That works.
* “a static method cannot be overridden”; **you cannot override a static method with a non-static method; however, you can override a static method with another static method**. You can call the overridden static method using SuperClassName.staticMethodName();
  + You do inherit a static method but overriding it isn’t “overriding” it’s “hiding” the superclasses static method. Overhidding not overriding.
* **YOU CANNOT OVERRIDE A METHOD AND MAKE IT MORE PRIVATE THAN BEFORE:**
  + “ERROR: attempting to assign weaker access priviledges; was public”

## 11.5 – Overriding vs. Overloading:

* Overriding: “To provide a new implementation for a method in the subclass.”
  + They are necessarily in different classes related by inheritance.
  + Same signature and return type.
* Use @Override to make sure it overrides. If it doesn’t, you get a compiler error, if it does, then it’ll work just fine. This is useful to make sure you don’t have typos in the method name and what not.
  + Okay, if you try to override a private, with a public or private or override a public with a private using @Override, then you get an error.
  + You also get an error for overriding a static method with another static or non-static.
* Overloading: “To define multiple methods with the same name but different signatures.”
  + Can be in the same class or another class related by inheritance.
  + Same name but different parameter list.
  + **Note:** When overloading with the same signature but different return type, I get an error: “return type String is not compatible with double”. This happens regardless of the types I seem to use.

## 11.6 – The Object Class and its toString() Method

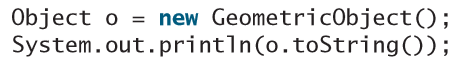
* Every class in java is descended from the java.lang.Object class.
  + If no inheritance is specified, then the superclass is always Object.
* By default, the object class’s toString() method returns a string with: the class name of which the object is an instance, @, the object’s memory address in hexadecimal.
* Calling System.out.println(object); on an object (of any class) is equivalent to calling System.out.println(object.toString());

## 11.7 – Polymorphism

* “Polymorphism means that a variable of a supertype can refer to a subtype object.”
* The three pillars of object oriented programming: encapsulation, inheritance, and polymorphism.
* A subclass is a specialization of every subclass; every subclass is an instance of it’s superclass but not vice versa.
  + E.g. every circle is a geometric object but…
  + You can always pass an instance of a subclass to a parameter of it’s superclass type.
* So if a method takes in a GeometricObject, you can pass in a Circle or a Rectangle.
* Polymorphism: An object of a subclass can be used whenever it’s superclass object is used! You can always just substitute a subclass object!
  + “A variable of a supertype can refer to a subtype object.

## 11.8 – Dynamic Binding

* “A method can be implemented in several classes along the inheritance chain. The Java Virtual Machine (JVM) decides at runtime which method is invoked.”



* **Declared type:** The type that declares a variable (the left side of the equal sign), in the above example, the type of o is Object.
  + It can hold a null value or a reference to an instance of the declared type.
  + “The instance may be created using the constructor of the declared type or its subtype.”
* **Actual type:** The actual class of the object referenced by the variable. In the example above, the actual type of o is GeometricObject.
  + The “toString();” method invoked is determined by o’s actual type. This is *dynamic binding.*
  + It is bound by the dynamic type.
* When a method is invoked on an object of a subclass, it will first search that subclass for the method, then it’s superclass, then the latter’s superclass, etc. until it finds such a method. It stops the search as soon as it finds it and uses the first one it finds.
* Listing 11.6 is an interesting example. They pretty much create three classes that extend each other (the upper one extend from Object) and override toString() in the top and middle one. Then they call toString() on that object to see which one is called. It works as described above but the top one (that extends Object) prints something weird. Remember that trying to print out something on an object with actual type object prints out “*java.lang.Object@130c19b*”
* **Matching a method signature vs. binding a method implementation:**
  + **Matching:** based on declared type, at compile time compiler finds a matching method according to parameter types, number of parameters and order of parameters.
  + **Binding:** A method may be implemented at several points along the inheritance chain. The JVM binds implementation of the method at runtime, decided by the actual type of the variable.

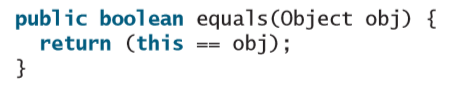
## 11.9 – Casting Objects and the instanceof Operator

* If method m takes in Object x: “public static void m(Object x) { … }“ then calling “m(new Student());” is equivalent to: “Object o = new Student(); m(o);”
* The statement below is known as *implicit casting*. It is legal because an instance of Student is an instance of Object:
* However, you couldn’t do the following because Object is not necessarily an instance of student:
  + This would cause a COMPILER ERROR, **because at compile time, o is of class Object!**
  + If you want to tell the compiler that o is of class student, you have to *explicit cast*:
  + Literally just write the Object o = new Student() line and this last line and it works just fine. Both compiles and runs!
* You can always *upcast*: cast an instance of a subclass to a variable of a superclass.
* To *downcast*: you should make sure (but technically it works without checking as long as it’s correct) that the Object is an instance of the subclass. If it’s not, you get a runtime exception called **ClassCastException.**
  + Just use an if statement:
* Casting is important because, if your code tries to call “o.aMethodInStudents()” without having cast it, you get an error. This is because, at compile time, since o is of class object, the compiler doesn’t see that it has a MethodInStudents().
* Also remember to use your brackets correctly:

* **Important Note:** Casting does not create a new object! In the example above, reference variables b and o point to the same object.
* **CASTING ERRORS ARE RUNTIME ERRORS!**
* ……

## 11.10 – The Object’s equals Method:

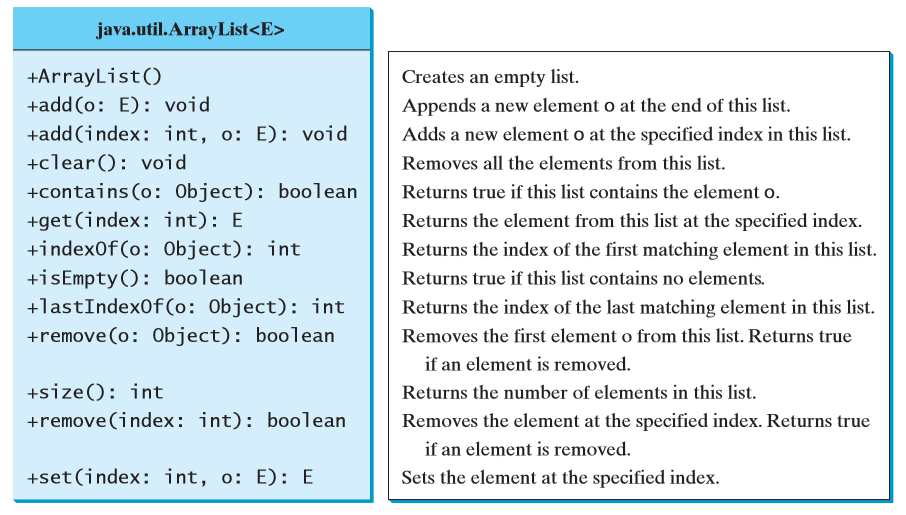
* The equals method is defined in the Object class (like the toString() method).
* This is what its implementation is:



* It just checks if the two reference variables point to the same object.
  + You should override it to test for content.
* **Note on comparing with ==: TESTS REFERENCE** If you create (initialize) two new objects with identical parameters then you will get false when using the default equals method, they point to different objects even if those are identical. BUT if you create two ints for examples with an identical number, that returns true because they are not objects. (If you create two new Integers with the same number, you also get false).
* **Note on comparing with .equals(): TESTS CONTENT** So the default equals method just uses == and will do what’s written in that last point. But for some classes (String, Integer, etc.) it has been overridden and will check the content, as you should if you override the equals() method. **The == operator is “stronger” than the .equals() method because it tests if the two variables reference the same object.**
* **IF YOU OVERRIDE THE EQUALS METHOD YOU WILL GET A CHECKSTYLE ERROR IF YOU DON’T ALSO OVERRIDE HASHCODE! (public int hashcode() {…})**
  + haschode() returns an int.
* WHEN YOU OVERRIDE THE METHOD, DON’T SET THE INPUT PARAMETER AS A SUBCLASS. YOU DON’T KNOW WHAT SORT OF OBJECT THESE SILLY KIDS WILL PASS IN. IT SHOULD ALWAYS BE:
  + And then check instanceof inside the code:
* THIS IS ALSO TRUE BECAUSE THE ORIGINAL EQUALS METHOD TAKES IN AN OBJECT SO YOU’RE NOT OVERRIDING IT IF YOU TAKE IN ANYTHING OTHER THAN AN OBJECT. If it’s not overridden, it will always use the default equals() method.

## 11.11 – The ArrayList Class:

* Use this instead of arrays to store unlimited number of objects, it also has a bunch of methods to make it easier to work with.
* YOU HAVE TO IMPORT ARRAYLIST: import java.util.ArrayList;
* ArrayList is a generic class! (Chapter 18 or 19 I think)
* Basically, the format to create an array list is this:
* But because of this cool thing called *type inference* you can just write this:
* Here are some of the methods you can use on the reference:
* Note: indexOf returns -1 if the object is not in the list.



* Note: The toString() method will return “[e1, e2, …]” where e1… are the elements in the array
* Note:
* **The elements in an array list CANNOT be of a primitive data type** (you can’t do ArrayList<int>)**, if you want to make an ArrayList of ints, do ArrayList<Integer> instead**
  + **Same with every other built – in data structure**
* You can sort an array list with: java.util.Collections.sort(referenceNameOfList);
  + To do the same with a regular array you would use java.util.Arrays.sort(referenceNameOfArray);

# Chapter 12: Exception Handling and Text I/O

12.10 – The File Class:

* The File class has the methods for manipulating files and directories
* Data stored in your program is lost when the program terminates.
  + To permanently store the data, you need to save it to a file
  + Files can be read and used by other programs
* An *absolute file name/full name* includes the name of the file and it’s complete path
* Skipped a bit
* The File class contains the methods for obtaining file and directory properties, for renaming files and for deleting them.
  + **The File class does not contain methods for reading and writing file contents.**
* **The File name is a string**
* new File("c:\\book") creates a file object for the directory c:\book
* Note that you have to use \\ and not \ to specify the path
* …

# Chapter 13: Abstract Classes and Interfaces

## 13. 1 – Introduction:

* An interface is to define common behavior for classes.

## 13.2 – Abstract Classes:

* **An abstract class cannot be used to create objects.**
  + It can contain abstract methods that are implemented in concrete subclasses
* Classes become more specific and concrete with each new subclass.
  + Sometimes a superclass is so abstract that it can’t be used to create specific instances.
* E.g. In the GeometricObject class, you can’t calculate the area and perimeter without defining the specific objects. But you can for a rectangle and circle.
* Add the word “abstract” to make abstract classes and methods.
* **Abstract classes are like regular classes, but you cannot create instances of abstract classes using the *new* operator**
  + They are implemented through subclasses
  + A class that contains abstract methods **must** be defined as abstract.
* The constructor in an abstract class can be **protected** because it is only used by subclasses.
* getArea() and getPerimeter() would be abstract in GeometricObjects because a “geometric object” doesn’t have an area or perimeter, but the implementations do
* Implement them and extend in subclasses as you usually do.
* **Note:** “You can create a reference of an abstract area class, but you can’t initialize it (new …())”

### 13.2.1 - Why abstract Methods?

* Basically, if you want to compare the area of the circle and rectangle, you would do something like circleObject.getArea() == rectangleObject.getArea().This wouldn’t work unless getArea() is defined in the superclass….???
  + UPON RE-READING THIS LATER, I HAVE NO IDEA WHAT THIS SECTION OF MY NOTES AND OF THE TEXTBOOK ARE TRYING TO SHOW

### 13.2.2 - Interesting Points about Abstract Classes

* **If a subclass of an abstract class does not implement all the abstract methods, it must also be defined as abstract.**
  + **Abstract methods are non-static** (I guess they’re dynamic because they are re-implemented differently in each subclass)
* You can’t instantiate an abstract class with the *new* operator but you can still define a constructor in an abstract class, which is invoked in the constructors of subclasses
* You can create an abstract class without any abstract methods. This would prevent the creation of objects of that class.
* You can override a non-abstract method to make it abstract. This is very unusual but is used if the application of the method isn’t valid in the subclass anymore. The subclass must now be abstract.
* A subclass can be abstract even if it’s superclass is concrete (e.g. Object is concrete but you can define abstract classes, which are by default extensions of Object).
* You can make an array of an abstract class:

## 13.5 – Interfaces:

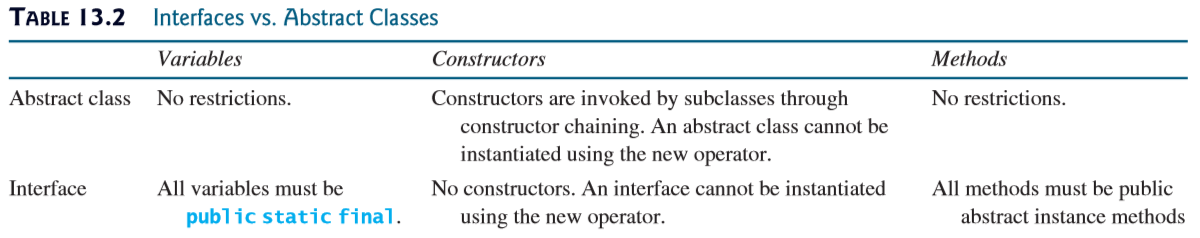
* Interfaces are similar to abstract classes but specify a common behavior for objects
* **An interface contains only constants and abstract methods.**
* To define an interface, write “interface” instead of “class” in the class header
* **You cannot create an instance of an interface using the *new* operator.**
* You use interfaces more or less the same way you use abstract classes.
* Use “implements” instead of “extends” to implement interfaces
  + Relationship is called *interface inheritance* (basically the same as regular inheritance)
* When a class implements an interface, it implements all the methods defined in it with exact signature and return type
* If a class implements an interface but does not implement its methods, it can be declared as abstract to avoid errors. The subclasses will then need to implement those methods.
* Listing 13.7 is a very good example!
* **In an interface, all data fields are** *public static final* **and all methods are** *public abstract.*

## 13.6 – The Comparable Interface

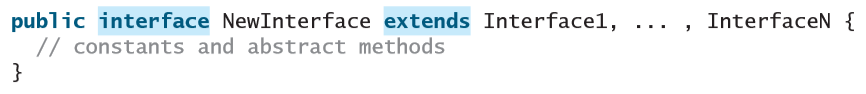
* The comparable interface defines the compareTo method for comparing objects.
* Anything that can be compared can implement it.
* It is a generic interface.
* Many classes in the Java library implement interface: Byte, Short, Integer, Double, String, Date, etc.
* To implement it, you just need to override the compareTo method.
  + **WHEN YOU IMPLEMENT IT IN THE CLASS HEADER, YOU HAVE TO SPECIFY THE TYPE. E.G. IMPLEMENTS COMPARABLE<INTEGER>**
* It will return a negative if the object before the dot operator is smaller than the object after the dot operator.
  + It returns 0 if they are equal.
  + You should be consistent with this when overriding.
* OBJECTS ARE instanceof THE INTERFACES THEY IMPLEMENT.
* The java.util.Arrays.sort(Object[]) in the Java API uses the compareTo method to sort arrays!!
  + Note: The one for string sorts in alphabetical order.
* You cannot compare objects of classes that don’t implement comparable.
  + But if it does implement comparable, then objects you create of that type will be comparable.
* It would be difficult to use a generic sort method without interfaces. Multiple inheritance would be necessary.
* The compareTo method is not implemented in the Object class (this is debatable wheter it should be) that’s why the Comparable interface exists.

## 13.8 – Interfaces vs. Abstract Classes

* A class can implement multiple interfaces but extend only one superclass
* Abstract classes can be used more or less the same way as interfaces but *defining* them is different:



* **An interface can inherit other interfaces *using the extends keyword***, this is called a *subinterface*:



* + **A class that implements NewInterface must implement the abstract methods in all the parent interfaces**
  + **An interface can thus extend other interfaces, but NOT classes**
* ..
* ..
* Interfaces are usually preferred to abstract classes because they can define a common supertype for unrelated classes
  + They give the example of defining an abstract Animal class that has a howToEat() method; you can define animal classes that extend this abstract class and implement how to eat, but they must all be animals, and edible (if class is concrete)
  + In comparison, you could define an Edible interface that is implemented not just by edible animals but also by vegetables, fruits, etc.

# Chapter 14: JavaFX Basics

14.1 – Introduction:

* Used to develop java GUI programs.
* Good example of object-oriented programming.

14.2 – JavaFX vs Swing and AWT

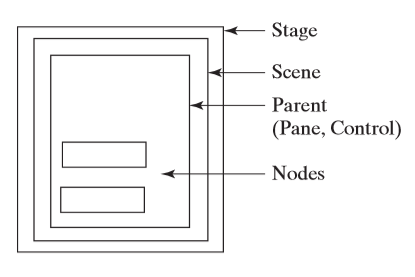
SKIPPED

14.3 – The Basic Structure of a JavaFX Program:

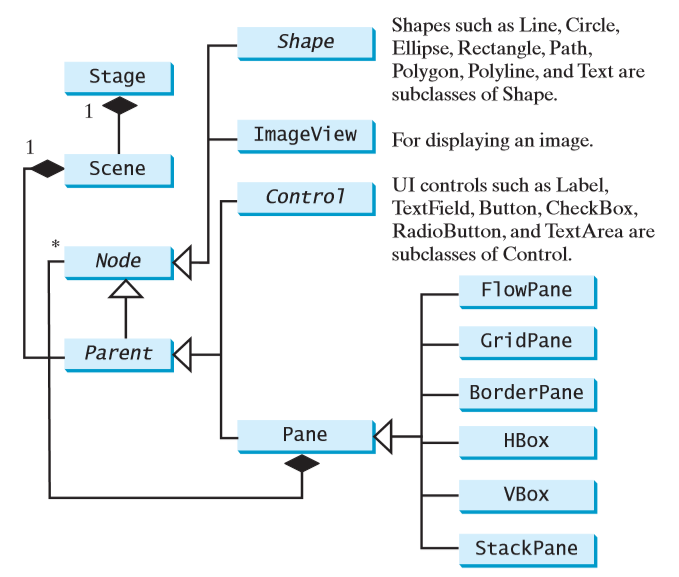
* javafx.application.Application is an abstract class that defines the framework for writing JavaFX programs.
* Every JavaFX program is defined in a class that extends javafx.application.Application
* **You have to import javafx.application.Application**
* **You have to import javafx.stage.Stage;**
* The launch() methd is a static method in Application class.
  + Call launch(args) in your main method. Don’t do anything else in your main method.
* **You must also override public void start(Stage primaryStage) {}**
* Button goes in a Scene, which goes in a Stage
* Create a scene with: Scene(node, width, height);
* A stage is a window, a primary one is automatically created when the application is launched.
* You have to: primaryStage.setScene(scene);
* primaryStage.show();
* You can create new stages the usual way: “Stage stage = new Stage()”
* You can prevent the user from resizing the stage: “stage.setResizable(false);”
* After you launch(), the JVM constructs an instance of the class using its no args constructor. Then it invokes the start method.

14.4 – Panes, UI Controls and Shapes

* Those three things are subtypes of Node
* You use nodes to help organize stuff the way you want it. You place nodes inside a pane and then a pane inside a scene.



* + A node is a visual component: a shape, image view, UI control or a pane
* A *shape* is text, a line, circle, ellipse, rectangle, etc.
* A *UI Control* is a label, button, check box, radio button, textField, etc.
* A Scene cannot contain a Shape of ImageView



* It can contain any subtype of node though
* Every subclass of node has a no args constructor
* The getChildren() method returns an instance of javafx.collection.ObservableList
  + Observable lists are a lot like Array lists
* Then you just use the add(e) method
* For shapes, or at least circles, you can setCenterX(); and setCenterY();
* **Note:** All measurements for height, width and stuff is in pixels

14.5 – Property Binding:

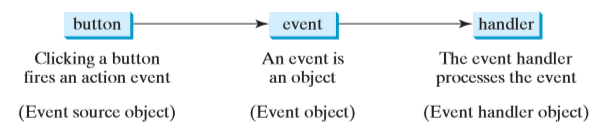
* You bind a target object (*binding object*) to a source object (*bindable/observable* object), a change in the source reflects in the target object.

**…**

# Chapter 15: Event – Driven Programming and Animations

## 15.1 – Introduction:

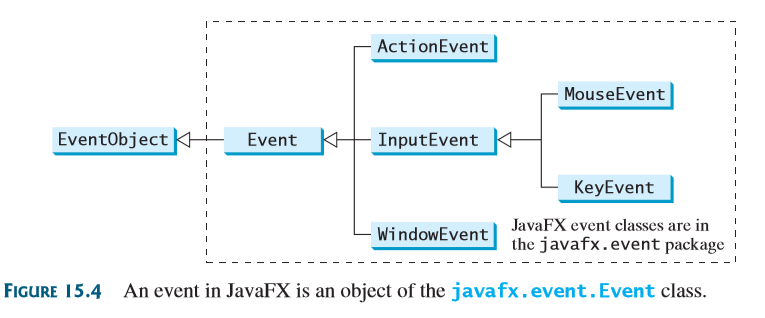
* You can process events like button clicks, mouse movement and keystrokes
* A button is an *event source object*: where the action originates
* You also need an object that can handle the action event on a button
  + An *event handler*



* To be a handler of an action event, an object must:
  + Be an instance of EventHandler<T extends event> interface.
  + The EventHandler object must be registered with an event source object using “source.setOnAction(handler)”;
* The EventHandler<ActionEvent> interface contains a handle(ActionEvent); method for processing the action event.
  + **You must override this handle(ActionEvent e) method in your HandlerClass (which must implement EventHandler<ActionEvent>)**
* You have to import
  + javafx.event.ActionEvent;
  + javafx.event.EventHandler;

## 15.2 – Events and Event Sources:

* A GUI interacts with the user, the events drive the execution. That’s why it’s called *event-driven programming*
* Events are triggered by external user action
* The component that creates an event is the *event source object*
  + A button is a source object, usually
* An event is an instance of an event class



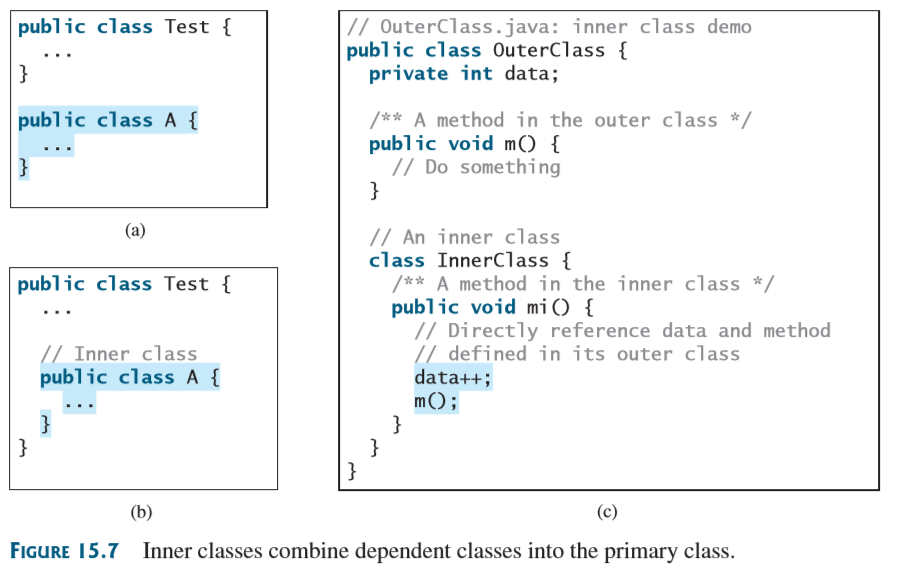
* You can identify the source object of an event object using **getSource()**
  + That method is in the EventObject class
* The ActionEvent is the event that is fired (could also be InputEvent or WindowEvent)
* If a component can fire an event, any subclass of that component can fire the same type of event
  + **E.g. Node can fire MouseEvent, KeyEvent and ActionEvent. So any of its subclasses (like button) can fire them as well.**

## 15.3 – Registering Handlers and Handling Events

* Java uses a *delegation* based model for event handling.
* An event handler is also called an event listener
* Again, for object to be a handler, two things are needed:
  + The event-handler object must be an instance of the corresponding event-handler interface. There is a unified interface EventHandler<T extends Event> which contains handle(T e)
  + The handler object must be registered by the source object. The method of registration actually depends on the event type. It is **setOnAction()** for an ActionEvent. For a mouse pressed event it is **setOnMousePressed()**, for a key pressed it is **setOnKeyPressed()**
* When you click the button, the Button object fires an ActionEvent, it passes it to the handle(ActionEvent) method to handle it.
  + The ActionEvent object contains information about the event that is obtained in the handle method.
  + E.g. You can use e.getSource() to obtain the source object that fired it.
* **Skipped a bit**

## 15.4 – Inner Classes:

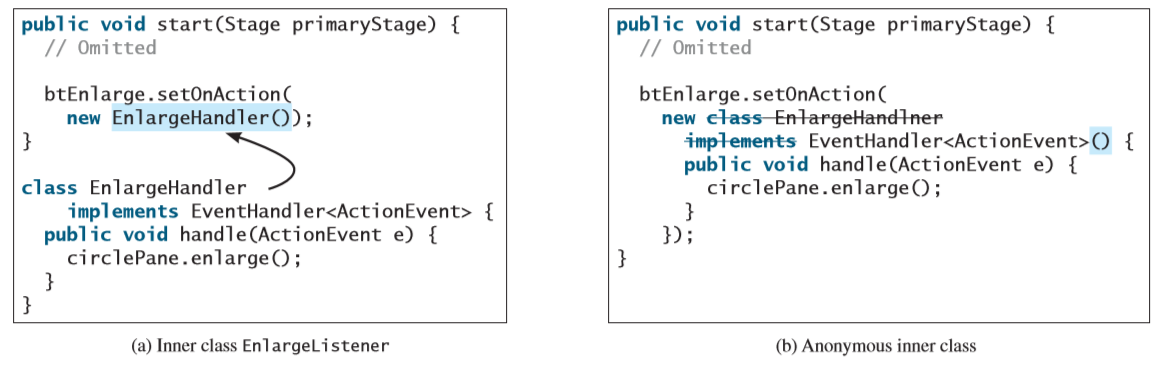
* An inner/nested class is defined within the scope of another class.



* Inner classes can be used just like regular classes.
* Usually, you define a class as an innder class if it is only used by its outer class.
* An inner class is compiled as **OuterClassName$*InnerClassname*.class**
* An inner class can access the data and methods defined in its outer class.
  + I tried it, if you add an instance variable to the outer class, then the methods of the inner class can use them!
* “An inner class can be defined with a visibility modifier subject to the same visibility rules applied to a member of the class.”
* An inner class can be defined as static.
  + A static inner class can be accessed using the outer class name
  + A static inner class cannot access non-static members of the outer class (just like how a static method cannot access non-static variables and methods)
* If you make your inner class static, then a method in there static you can make a call like: “outerClass.staticInnerClass.staticInnerClassMethod()”. No problem. So if the inner class and the method are static, it’s fine. But:
  + If you make the inner class not static (method still static) you get an error
  + If you make the inner class static and method non-static, you can’t call the non-static method that way!
* You can create objects of the inner class from external classes. If the inner class is nonstatic, you must first create an instance of the outer class (outerObject) and then use this format:
  + OuterClass.InnerClass innerObject = outerObject.new InnerClass();
* If the inner class is static:
  + OuterClass.InnerClass innerObject = new OuterClass.InnerClass();
* A simple use of inner classes is to make class files easier to organize since they are named with the outerclass as prefix.
* It can also help you avoid class naming conflict.
* A handler class is designed specifically to make a handler object for a GUI component. So it will not be used by other applications, which is why it makes sense to make it an inner class.

## 15.5 – Anonymous Inner Class Handlers

* An anonymous inner class is just an inner class without a name.



* + Combines defining an inner class and creating an instance of the class into one step.
* Basically, you use the following format I think:

buttonName.setOnAction(new EventHandler<ActionEvent>() {

public void handle(ActionEvent e) {  
//Whatever you want to do

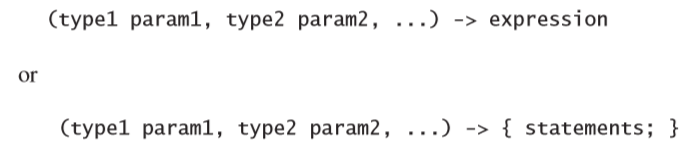
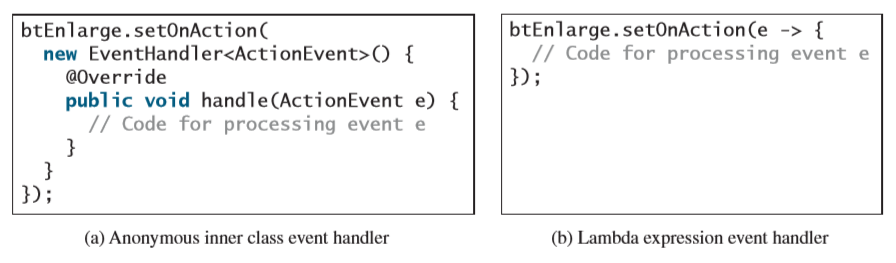
}

});

* Anonymous inner classes are just a special kind of inner classes but have the following features:
  + The anonymous inner class must always extend or implement a superclass/interface but does not have an explicit extends/implements called. In the example above, the EventHandler<ActionEvent> is the interface it implements.
  + You must implement all the abstract method of the superclass/interface
  + An anonymous inner class always uses the no args constructor form its superclass (that’s why we add () after the <>) to create an instance.
  + Anonymous innerclasses are compiled as **OuterClassName$n.class** where n is an integer representing the number of the inner class.

## 15.6 – Simplifying Event Handling Using Lambda Expressions:

* Lambda expressions are some pretty crazy shit:
* The basic syntax for a lambda expression can be either:



* The data type for the parameter can be explicity declared (ActionEvent) or inferred by the compiler.
* The parentheses can be omitted if there is only one parameter without an explicit data type. (If it has a data type, you need the parentheses)

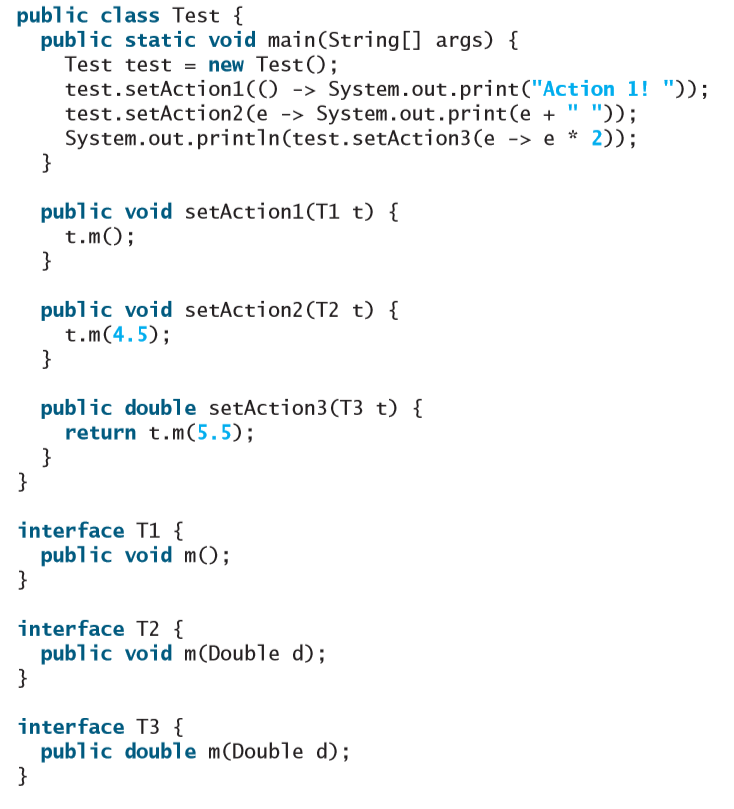
e -> {

//Code for processing e

}

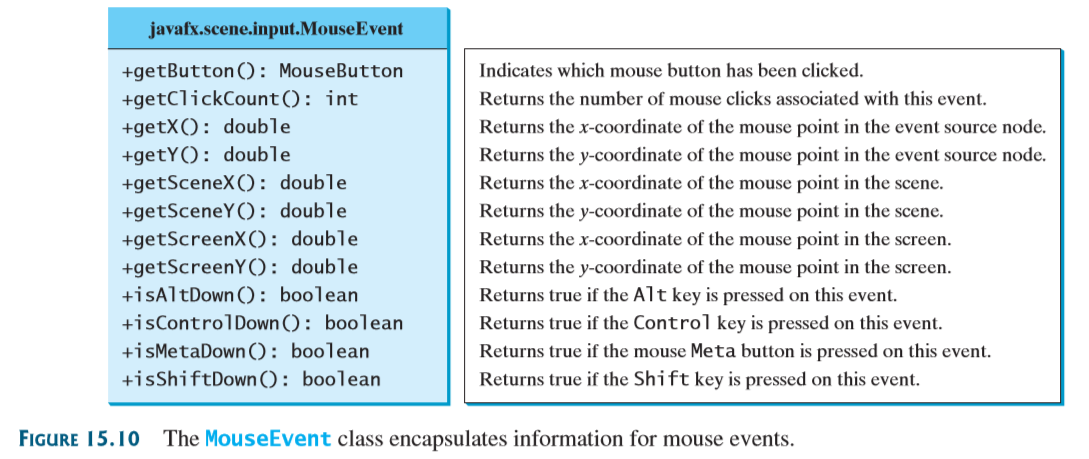
* The compiler treats a lambda expression as an object created from an anonymous inner class.
  + In this case, the compiler understands that e must be an instance of EventHandler<ActionEvent>. Since the EventHandler interface only has one method, the handle() method, and since it only uses an ActionEvent type, then it knows that e is of that type and the content is for the handle() method.
  + If the interface contained more methods, it wouldn’t be able to compile the lambda value.
  + The interface must contain only one abstract method.
  + This is called a *functional interface* or a *Single Abstract Method* (SAM)
  + You can also even omit the braces: {} if there is onlye one statement: buttonname.setOnAction(e -> System.out.println(“this works!”));
* **You can actually define your own setAction() methods!! But you also have to write the functional interfaces to go with it.**
* If you try to setOnAction more than once, it will only do the last one that you setOnAction.

**Crazy example:**



## 15.8 – Mouse Events:

* A **MouseEvent** is fired whenever a mouse button is pressed, released, clicked, moved or dragged on a node or a scene.
* The MouseEvent object the number of clicks, x and y location of the mouse, which mouse button was pressed, etc.
* Four constants: PRIMARY, SECONDARY, MIDDLE and NONE are defined in MouseButton to indicate which button was pressed if any.



* + Use getButton() to find out. MouseButton.PRIMARY/SECONDARY/MIDDLE/NONE
* Each node or scene can fire mouse events.
* The following are tall the methods to register a handler for different things:
  + setOnMouseClicked(handler)
  + setOnMousePressed(handler)
  + setOnMouseReleased(handler)
  + setOnMouseEntered(handler)
  + setOnMouseExited(handler)
  + setOnMouseDragged(handler)
  + setOnMouseMoved(handler)

## 15.9 – Key Events:

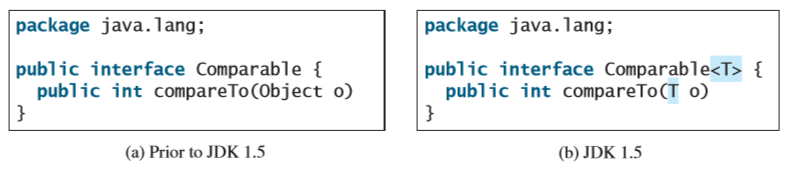
# Chapter 19: Generics

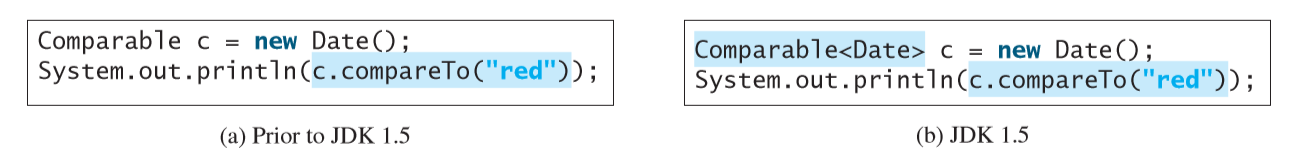
## 19.1 – Introduction:

* ArrayList is an example of a generic class.
* Comparable is an example of a generic interface.
* Used on classes and methods
* Generics let you *parametrize* types: you define a generic type at the start of your class or method and, at compile time, the compiler will substitute that with a concrete type.
  + In ArrayList, you can store Strings, Integers, etc. because it defined a generic type that can be substituted.
* Key benefit is that generics enable errors to be detected at compile time rather than runtime
  + If you attempt to use an incompatible object type, you get a compiler error (after you specify the type in your declaration I guess)

## 19.2 – Motivations and Benefits:

* This is how the Comparable method was written before the implementation of Generics.
* T is the formal generic type that is then replaced by an actual concrete type (g*eneric instantiation*)



* We usually use E or T.
* Look at the two codes below for an example of how it’s useful (try to see the error):
  + The first one declares c to be Comparable and calls compareTo
    - It would compile just fine but would cause a runtime error because you can’t compare a Date and a String
* If you write an ArrayList as:
* Then you can only add strings to list:
* Something like this returns a compiler error:
* **You can’t replace a generic type with a primitive data type**
  + So you couldn’t do:
  + But you could do:
* Then, when you intList.add(5); for example, it will *autobox* 5 into “new Integer(5);”
  + Because Integer is the wrapper type class.
* You don’t need to cast anything when retrieving it, the compiler knows the element type.
* Ok so here’s the thing: When you declare an object (such as an ArrayList object) with a specific concrete type, it will only accept input objects of that SPECIFIC type AND its subclasses.
  + This is true UNLESS it’s something like Integer or Double which has an equivalent primitive data type, then you can add the primitive data type.
  + Also list.get(); for example, will retrieve it of the class declared in <>. So you could do <Integer> and add a bunch of ints (the only things you could add are Integers and ints) and you can do Double d = list.get(2); if the list is a list of Integers! **Autounboxing**

## 19.3 – Defining Generic Classes and Interfaces:

* You can define a generic type for a class or interface.
  + A concrete type must be specified when using the class/interface to declare instance variables though

…

## 19.4 – Generic Methods:

* You can declare generic methods in non-generic classes but the format to use them is a bit different.
* Of course you can replace void with a specific return type
* Print is just an example method name
* If the method is in class GenericMethodDemo, use print like this: GenericMethodDemo.<Integer>print(integers); or just “print(integers)”
  + Could be different types

…

19.7 – Wildcard Generic Types:

* You can use unbounded wildcards, bounded wildcards or lower-bound wildcards to specify a range for a generic type.
* Inside your <> you can put:
  + ?
  + ? extends T
  + ? super T

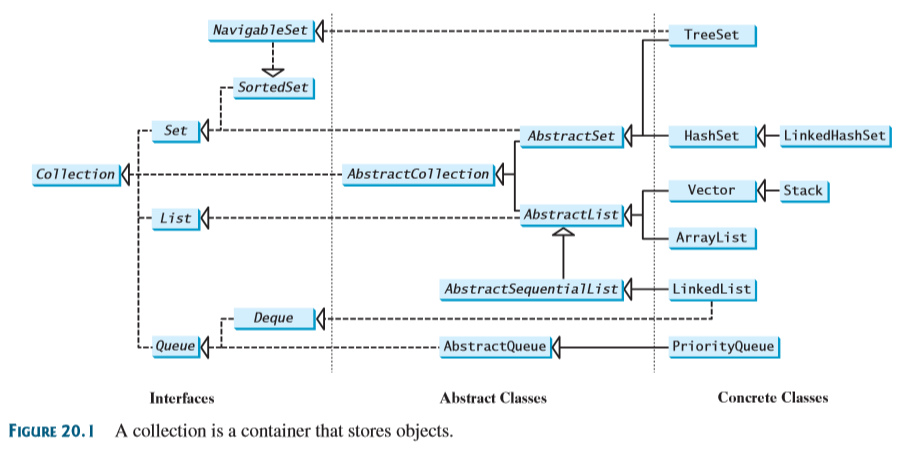
# Chapter 20: Lists, Stacks, Queues, and Priority Queues

## 20.1 – Introduction:

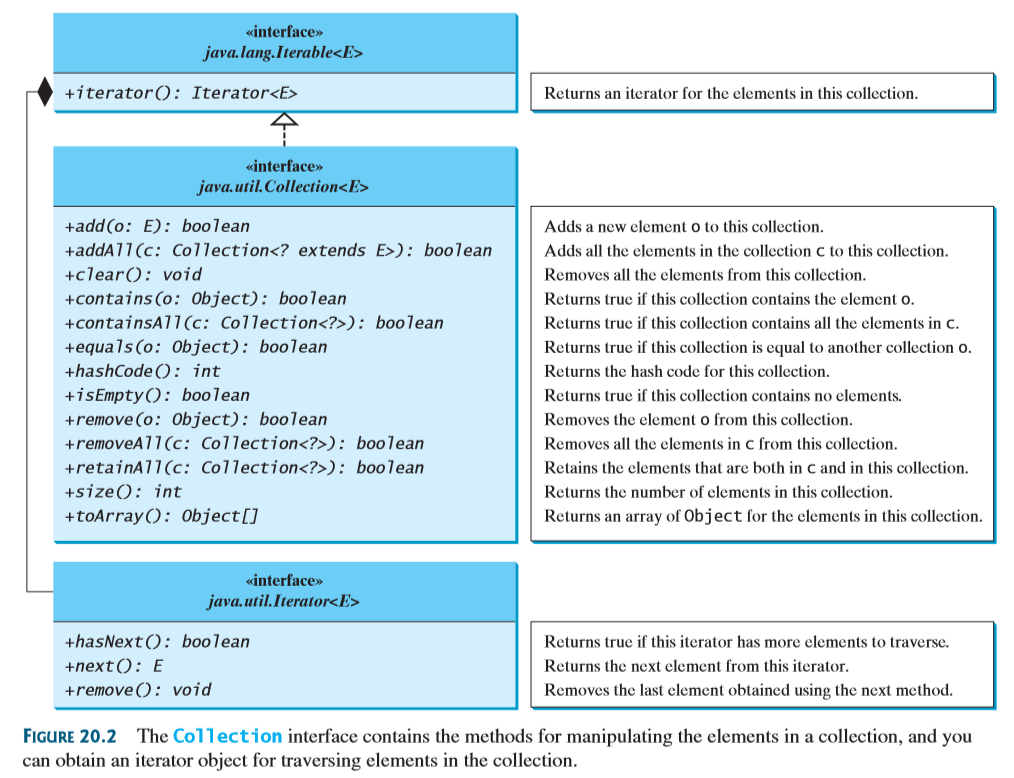
* A *data structure* is a collection of data organized in some fashion
  + It also supports operations for accessing and manipulating the data.
* It is also known as a *container* or *container object*
  + Stores other objects, referred to as *data* or *elements*
* “Defining a data structure is essentially defining a class”
  + It’s really defining an instance of a class.
  + Should store data and provide methods to support operations.
* ArrayList is an example of a data structure
  + There are more, known as the *Java Collections Framework*

## 20.2 – Collections:

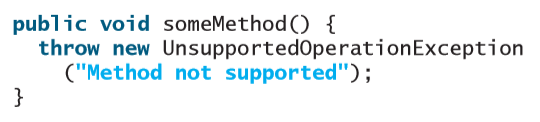
* There is a **Collection** interface
* The Java Collections Framework support two types of containers:
  + A *Collection:* For storing a collection of elements
  + A *map:* For storing key/value pairs
* Examples of Collections:
  + **Set**s: A group of non-duplicate elements
  + **List**s: Store an ordered collection of elements
  + **Stack**s: Store objects that are processed in a last-in, first out fashion.
  + **Queue**s: Store objects that are processed in a first-in, first-out fashion.
  + **PriorityQueue**s: Store objects processed in order of their priorities
* These collections are defined in interfaces (along with their common features)



* **All the interfaces and classes defined in the Java Collections Framework are grouped in the** java.util package
* This tree is pretty cool because you can just extend from the abstract classes you want and only implement some of the things from Collection. So you don’t have to implement everything.
  + For example: the AbstractCollection class implements all the methods in Collection except the add, size and iterator methods.
  + Which are implemented in subclasses
* The Collection interface provides a bunch of basic operations:



* Some of these methods cannot be implemented in concrete subclasses (I think this is in the sense that they don’t really have any meaning, you can implement them but it doesn’t make sense in the context of your collection).
  + **So then you just make the method throw a java.lang.UnsupportedOperationException (which extends from RuntimeException)**
  + Basically, you implement it as follows



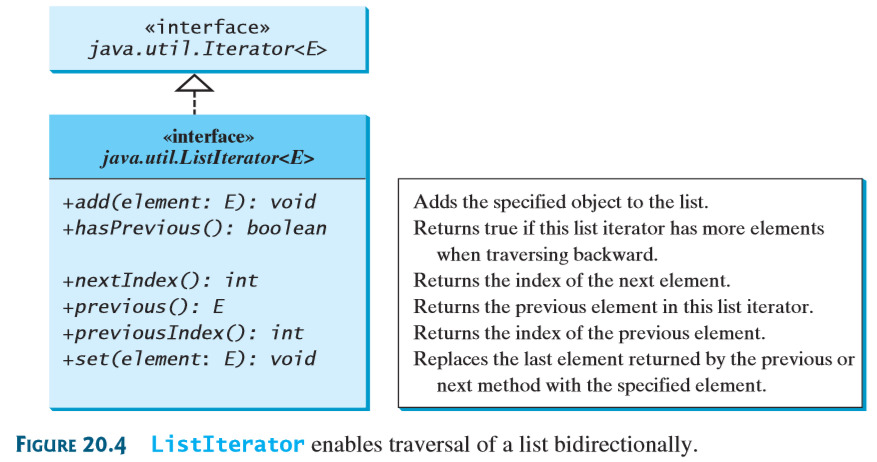
* They have an example here that shows the use of multiple methods, it’s pretty good
  + But basically just understand the methods above

## 20.3 – Iterators:

* Every collection is **Iterable**: You can obtain its iterator object to traverse all the elements in a collection.
* Iterators are useful for walking through a data structure without exposing how the data is stored in the structure.
* **Collection interface extends the Iterable interface**
* The Iterable interface defines an iterator method, which returns an iterator.
  + iterator() in the Iterable interface returns an instance of Iterator.
  + You can call .next(), .hasNext() and .remove() on the iterator
* **WHEN YOU CREATE THE ITERATOR, YOU DON’T DO “Iterator<type> iterator = new Iterator();**
  + **INTEAD, YOU DO: “Iterator<type> iterator = collection.iterator()”** where collection is the name of your collection
  + **You’re basically calling the iterator method on your collection**
* Note: you can’t print out the iterator itself, it will give you a memory address. Print out “iterator.next()” instead.
* Remember that the class of the iterator is “Iterator<type>” so you can’t do: “iterator = iterator.next();” because that will return a type and try to set it equal to an Iterator<type>.
* Note that you can also just use a foreach loop to iterate through the collection.

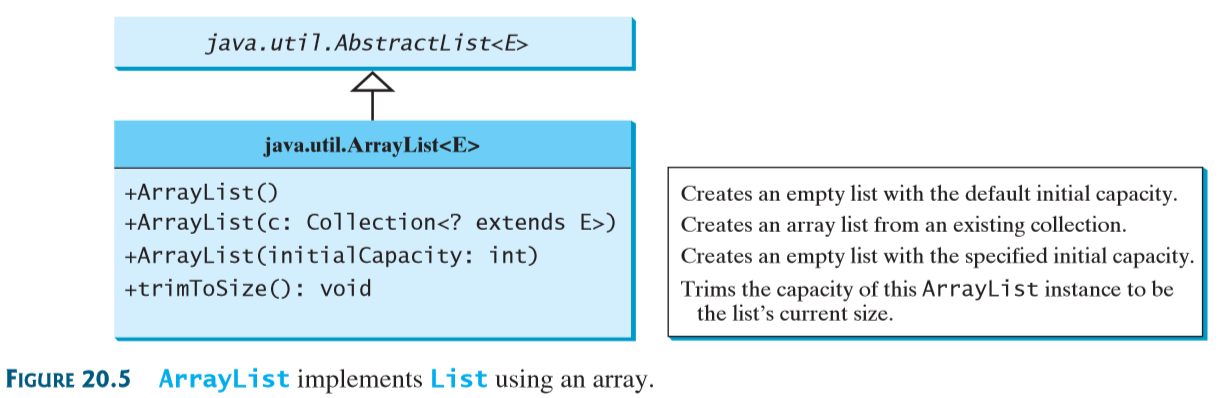
## 20.4 – Lists:

* The List interface extends the collection interface. Use
  + ArrayList
  + LinkedList
* Stores in sequential order.
* The List interface allows duplicates, it also adds position oriented operations. For example:
  + add(index, element);
  + addAll(index, collection); //Adds the whole collection at specified index
  + remove(index);
  + set(index, element); //Replaces the element at that index
  + indexOf(element); //Returns the index of the element’s first instance.
  + lastIndexOf(element);
  + subList(fromIndex, toIndex);
* Also adds a new iterator that can traverse list bidirectionally.
* There ListIterator interface extends the Iterator interface to give extra capabilities for the Iterator:

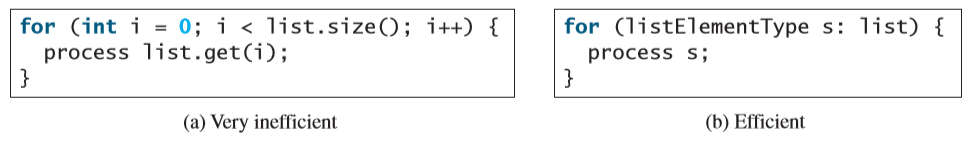


### 20.4.2 – The ArrayList and LinkedList Classes

* Both are concrete implementations of the list interface
* ArrayList stores elements in an array
  + If you want random access to indices without inserting or removing elements at the beginning of the list.
  + ArrayList is somewhat resizable, the array has an initial size capacity but will grow if more elements are added.
  + It will not automatically shrink, use trimToSize()



* LinkedList stores elements in a linked list
  + Lists can grow or shrink dynamically, arrays have a fixed lengths once created.
* There’s a pretty good example but, basically, ArrayList and Linked List operate similarly.
* The difference lies in their implementation and consequent performance.
  + ArrayList is efficient for retrieving elements and for adding or removing them at the end of the list.
  + Linked list is efficient for inserting and removing elements anywhere in the list.
  + They have the same performance for adding and removing elements in the middle and end of the list
* There is a get(i) method for LinkedList but it is inefficient:



* If you just have a bunch of words or ints and you want to make a string out of them, you can do that right away using: Array.asList(arrayObject);

# Chapter 21: Sets and Maps

## 21.1 – Introduction:

* A set is a data structure to store and process **non-duplicate elements**.
* A map provides a way of quickly looking up and retrieving a value used in a key.

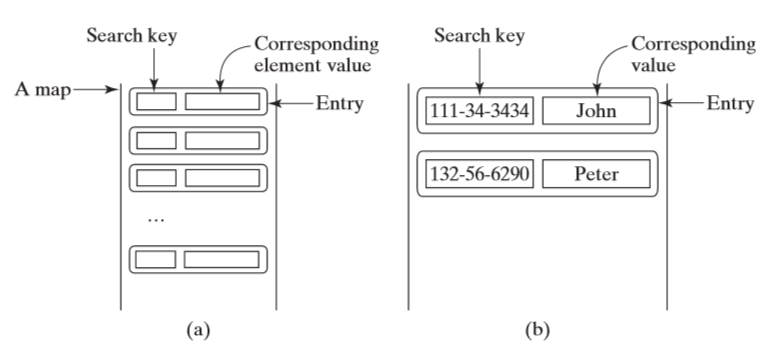
## 21.2 – Sets:

* There are three classes you can use to make sets
  + HashSet
  + LinkedHashSet
  + TreeSet
* Set interface extends Collection interface
  + It does not introduce new methods or constants
  + It makes it so that all elements of a Set are unique
* Concrete classes that implement Set must make sure that no duplicate elements can be added to the set.

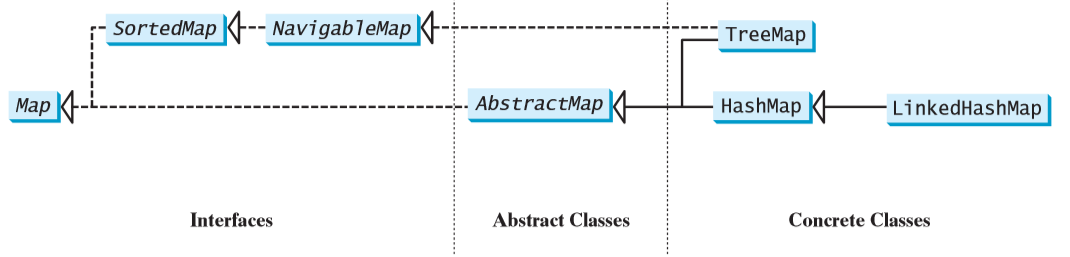
…

## 21.5 – Maps:

* You can create a map using one of its three concrete classes: HashMap, LinkedHashMat, TreeMap
* A map is a container object that stores key/value pairs
  + Enables fast retrieval, deletion and updating
  + All done through the key
* The keys act as indexes (instead of having integers like for lists, the key can be any object).
  + There can be no duplicate key.
  + Each key only maps to one value



* The Map hierarchy:



* The map interface provides most of the methods
  + I haven’t included a pic like I usually do so go check the api you lazy
* There are these things called ***update methods*:**
  + clear();
  + put(key, value);
  + putAll(map);
  + etc.
* And these things called **query methods:**
  + containsKey(key);
  + isEmpty();
  + etc.

**Skipped the rest**

# Chapter 22: Developing Efficient Algorithms

## 22.1 – Introduction:

* Algorithm design is to develop a mathematical process for solving a problem.
* Algorithm analysis is to predict the performance of an algorithm.

## 22.2 – Measuring Algorithm Efficiency Using Big O Notation:

* Big O notation gives a function for measuring algorithm time complexity based on the input size.
* It is difficult to compare algorithm efficiencies by comparing execution times because the time depends on several factors:
  + “Many tasks run concurrently on a computer…”
  + It depends on specific input
  + That’s why we developed big O
* The Big O approach looks at growth rate: the rate of increase of execution time as the size of the input increases.
* Linear search:
  + Compares the key with every element in the array sequentially.
  + If element is not in the array, it requires n comparisons.
  + If it is in the array, it requires n/2 on average.
  + Doubling the size of the array doubles the search time; the growth rate is n, linear.
  + O(n) – linear algorithm
* Execution time may vary based on input:
  + Best-case input
  + Worst-case input: The algorithm will never be slower than that
  + Average – case analysis: tries to determine the average execution time for all possible inputs of the same size. Difficult to perform. Worst case analysis is usually easier so done instead.
* When looking at the algorithm, ignore coefficients, they have marginal impact relative to the order of magnitude: O(n) = O(n/2) = O(100n)
* If the execution time is not related to input size, the algorithm is said to take constant time: O(1)
  + E.g. A method that retrieves an element at a given index in an array.
* You can also analyze space complexity: the amount of memory space used by an algorithm. It is O(n).

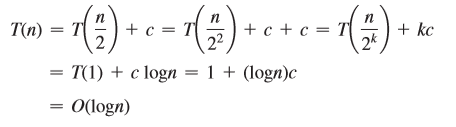
## 22.3 – Examples: Determining Big O:

* Just look through the examples in the book, they’re very useful.
* A simple for loop from 1-n usually has O(n)
* A nested for loop, both from 1-n has O(n^2)
* A nested for loop, one from 1-n and the other from 1-i has O(n^2)
* A nested for loop, one from 1-n and the other from 1-c has O(n)

## 22.4 – Analyzing Algorithm Complexity:

Binary Search:

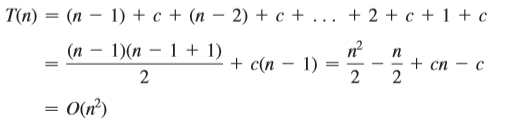
* TO DO BINARY SEARCH, YOUR ELEMENT MUST BE SORTED!
* Right, so say there are n elements in the array. Let’s say, for argument’s sake, that n is a power of 2. So n = 2^k. If you simplify that you get:
* Well every time, the binary search will split the array in half and just continue the search through one after a simple comparative operation, so the search time becomes T(n/2) for the new one, it then repeats. It does this until there is only one element left: T(1).



* Remember that k = log(n) – you ignore the base the same way you ignore coefficients – so you do that substitution and get the result above.
* **Binary search is O(log(n))**
* It is a *logarithmic algorithm* – logarithmic growth rate.
  + Each time you double the size of the array, you only add one comparison.
  + If you square the size of the array, you only double the execution time

Selection sort:

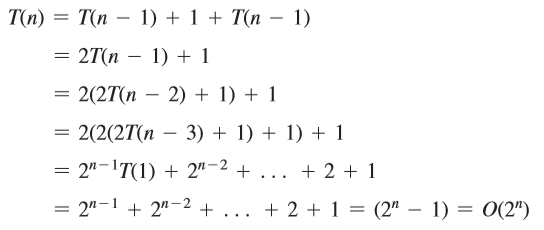
* It finds the smallest element in the list and swaps it with the first element. Then repeats until only one element left. First search is n-1, second search is n-2, etc.
* Btw, c represents the total number of other operations: assignments and other comparisons. So basically, each time you have (n – 1) + c and then (n – 2) + c, etc. until you only have one element left.



* Now look children, the -1 cancels with the 1, the -2 with the 2, etc. So originally, you had n – 1 occurences, but because it cancels out you have (n – 1)/2 occurences and those occurences are just ns.

Tower of Hanoi Problem:

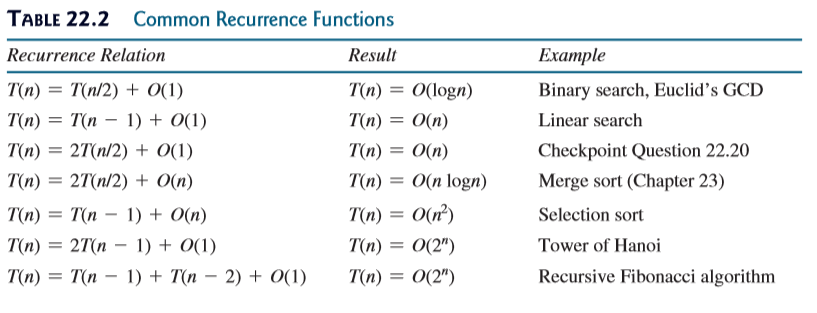
* The problem is just moving to recursively move n disks from tower A to tower B with the assistance of tower C.
  1. Move the first n – 1 disks from A to C with the assistance of tower B.
  2. Move disk number n from A to B.
  3. Move n – 1 disks from C to B with the assistance of tower A.
* Steps 1 and 3 both take T(n – 1) operations, step 2 just takes 1.
* But T(n – 1) operations is basically just 2T(n – 2) + 1 because, to solve for T(n – 1) you need to go through steps 1 – 3 again. And again for T(n – 2), you repeat steps 1-3 but for T(n -3), etc.



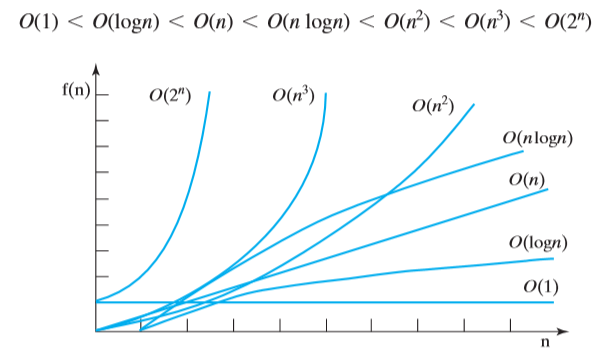
* **The tower of Hanoi problem has complexity O(2^n)**
* It is an *exponential algorithm* – exponential growth rate.
* As input size increases, execution time increases exponentially.
  1. Moving the disks at a rate of 1 per second would take 136 years to move 32 disks.
  2. 585 billion years to move 64 disks.

Common Recurrence relations:

* Just some common formulas:



Comparing Common Growth Functions:



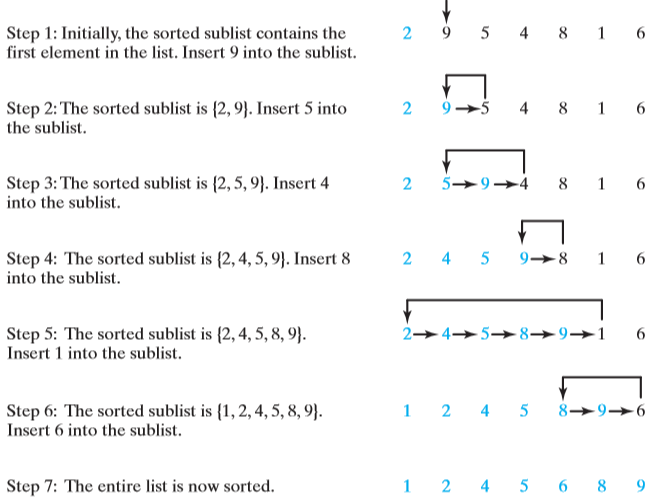
# Chapter 23: Sorting

## 23.1 – Introduction:

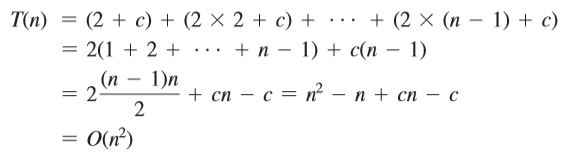
* Data to be sorted could be integers, doubles, characters or objects.
* This chapter assumes:
  + Data to be sorted is integers
  + Data are sorted in an array
  + Data are sorted in ascending order.

## 23.2 – Insertion Sort:

* So you take the next element and add it to a sorted array.
* You do this by saving the new element to insert in a temporary variable, comparing it to the elements starting from the highest one and slide all the ones that are greater up. You then insert it where appropriate.



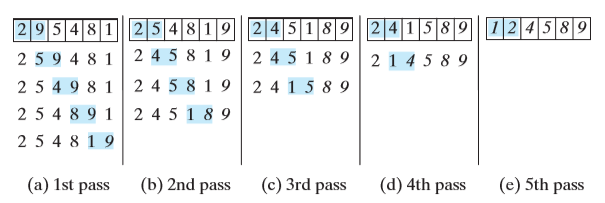
* Implementing it requires a nested for loop, one to add each element in the original array into the sorted array and one to compare the added element to all the ones already in the sorted array, to slide them up appropriately and to insert the added element.
* I don’t fully understand the analysis but:



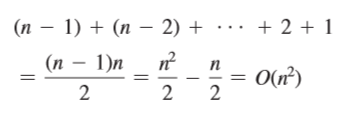
* **Best case: O(n)**
  + This is because it will go through each element, but upon comparing the new one with the highest in the sorted array, it will see that it’s already in order so it won’t go through the inner for loop.
* **Worst case: O(n^2)**

## 23.3 – Bubble Sort

* It makes several passes through the array, each time it compares neighboring pairs and swaps them if the pair is in decreasing order.
  + That way, the highest value is bubbled to the top, then the second highest to n – 1, etc.



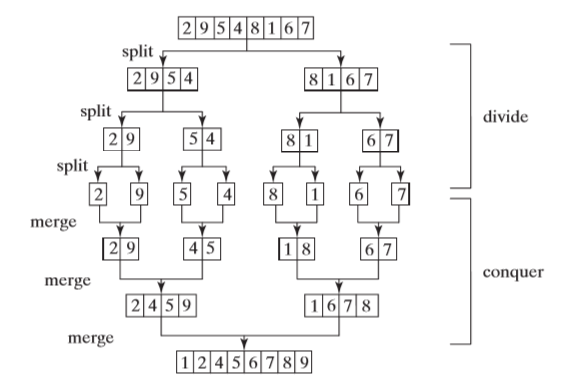
* Note that if no swap is made in a pass, there is no need to perform the next pass because all the elements are already sorted. You can use this to improve the algorithm.



* **Best case: O(n)**
  + This is because no swaps in first pass so no need to go through again.
* **Worst case: O(n^2)**

## 23.4 – Merge Sort:

* It recursively splits the array into twos until only one element is left in each, then it joins them one after the other.



# Chapter 24: Implementing Lists, Stacks, Queues and Priority Queues

## Chapter 24.1 – Introduction:

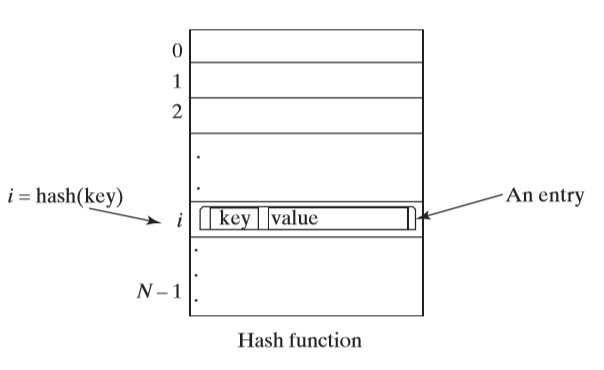
# Chapter 27: Hashing

## 27.1 – Introduction:

* Hashing is superefficient, it takes O(1) to search, insert, and delete an element using hashing.

## 27.2 – What is Hashing?

* Hashing uses a function to map a key to an index.
* Remember that maps are data structures that are implemented using hashing.
  + Each entry has a key and a value.
  + Keys are used to search for the corresponding value.
  + **A map is also called a dictionary, a hash table or an associative array.**
* If you know the index of an element in the array, you can retrieve it in O(1) time.
* The array that stores the values is called a *hash table*.



* The function that maps a key to an index in the hash table is called a *hash function*.
* Basically, a hash function takes the key and gives you an index from it.
* Ideally, each search key would map to a different index in the hashtable (*perfect has function*).
  + This is difficult
  + If two or more keys are mapped to the same hash value, a *collision* has occurred.

## 27.3 – Has Functions and Hash Codes:

* A hash function typically converts a search key to an integer value called a hash code, then it compresses the hash code into an index for the hash table.
* The Object class has the hashCode() method, it returns an integer hash code.
  + By default, it returns the memory address of the object. (If you just try to print randomObject.hashCode(), it will actually return some large integer)
* The contract for the hashcode method:
  1. Override hashCode method whenever the equals method is overridden to ensure that two equal objects return the same hashcode.
  2. During program execution, if the object’s data is not changed, invoking the hashCode method will return the same integer.
  3. Two unequal objects may have the same hash code but you should implement the hashCode method to avoid too many such cases.

### 27.3.1 – Hash Codes for Primitive Types:

### 27.3.2 – Hash Codes for Strings:

* An intuitive approach may be to sum the Unicode for all the characters as the hash code. This can result in a lot of collision if same letters: “dot” and “tod”
* A better approach takes into account the position of the characters into consideration.
* Si is s.charAt(i)
* Called a *polynomial hash code*
* You choose a value of b, good ones have experimentally been shown to be 31, 33, 37, 39 and 41.
* In the String class it is overridden using b as 33

### 27.3.3 – Compressing Hash Codes:

* The hash code can be very large and out of the hash-table index, so you need to scale it down.
* Assume the index for a hash table is beween 0 and N – 1, the most common way to scale an integer to between 0 and N -1 is:
* Choose N to be a prime number greater than 2 to spread indices evenly.
* Skipped the rest

## 27.4 – Handling Collisions Using Open Addressing:

* A collision occurs when two keys are mapped to the same index in a hash table
* Two ways to handle this are:
  + Open addressing
  + Separate Chaining
* We talk about Open Addressing only here
* Open Addressing just finds an open location in the hash table using one of the three methods below

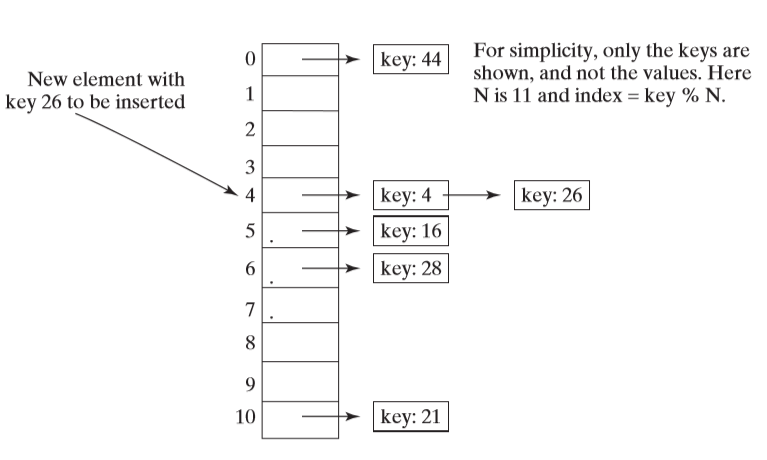
27.4.1 – Linear Probing:

* When a collision occurs at hashTable[k % N], check if hashTable[(k + 1) %N] is available, if not, check hashTable[(k + 2) %N], etc.
* Note: when probing reaches the end of the table, it goes back to the beginning.
  + Treated as circular
* To search for an entry in the hash table, obtain the index, say k from the hash function for the key. Check if hashTable[(k + 1) %N] contains the entry, if not, check hashTable[(k + 2) %N],etc. until the entry is found or an empty cell is reached
* To remove an entry from the hash table, search the entry that matches the key.
  + If found, place a marker on it to denote that the entry is available.
* Linear probing causes clusters that can slow down search time!

Skipped the rest

## 27.5 – Handling Collisions Using Separate Chaining:

* Separate Chaining places all entries with the same hash index in the same location, rather than finding new locations.



* + Each location uses a bucket to hold multiple entries
* You can implement it using an ArrayList or LinkedList.
* Each cell in the hash table is a reference to the head of a linked list. The elements are chained starting from the head.

## 27.6 – Load Factor and Rehashing:

* The load factor is the ratio of the number of elements to the size of the hash Table