CSC207 Cheat Sheet

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Week 1 Java Basic**  1**.The hybrid approach** (Java)**:** Before running, the human code is translated into bytecode (machine code) for the Java Virtual Machine. (Hardware independent)  2. **Pros of Static Typing**: More errors detected earlier in development, fewer errors at runtime and in shipped code.  3.**Pros of Dynamic Typing**: Deals naturally with certain types of self-describing data. Tends to reduce unnecessary clutter and duplication/repetition in code.  4.**Cons of Dynamic Typing**: More errors detected later in development and in maintenance. More errors at runtime and in shipped code.  5.**Primitive types**: byte (8-bit), short (16-bit), int (32-bit), long (64-bit), float (32-bit), double (64-bit), char (16-bit), boolean (1-bit)  Note：创建primitive type的值不需要用new, primitive type的值都是immutable的，variable指向的都是primitive type的value本身  6. **Class type**: all other types, String, Integer, Balloon，创建class variable时一定要用new这个keyword，variable指向的是这个object的address  7. **Auto-Conversion**: byte → short → int → long → float → double  char → int and above, boolean → no other types  8. 没有follow “can-auto-convert” directions must be explicitly casted. e.g. long x = 207; short y = (short)x;  8.**operators:** !(NOT),&&(AND),||(OR),^(XOR)  9.HashMap<Integer, String> hm = **new** HashMap<Integer,String>()  10.**Scope of variables**  -class scope: variables declared in a class with “static” keyword，class每个instance都share同一个  -class instance scope: variables declared in a class. 没有static keyword， 每个object都有一个自己的  -method scope: variables declared in a method, 出了这个method就用不了这个variable了  -block scope: variables declared in a method, and in a loop，在loop里的一个variable，出了loop就用不了，就算还在method，还是用不了  11**.Pass by value/reference**: 把一个variable放进function作为parameter时, primitive type放进去的是variable的值(并没有对原本这个variable产生什么影响)，class type放进去的是variable存的那个address，对parameter做什么改变的话是会找到那个address对address里的那个object对改变的。  12**. Static**: when used on method, method becomes function,不需要借助这个class的instance来call (static function里能含有non-static attribute/method)，当static used on variables时就像一个global variable，这个class的所有instance都share这同一个variable，并只会被initialize一次，,static method/attribute可直接通过class name调用。  13**. Compare things**  System.***out***.println(a == b);  // compares the reference  System.***out***.println(a.equals(b)); // compares the value  14. **private** Random rand = **new** Random();  **int** x = **this**.rand.nextInt(100); | **Week 2 OOP**  **Overloading**: 在一个constructor里call另一个constructor using this(parameter)  **Constructor in Child class**: call parent’s constructor by super(args). if don’t call, parent’s default constructor with no args will be called, i.e., super().   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Access modifiers** | class | package | subclass | world | | public | yes | yes | yes | yes | | protected | yes | yes | yes | No | | Default (pkg private) | yes | yes | No | No | | private | yes | No | No | No |   **Week 3 Junit，inheritance, UML**  **Juni**t:一个test suite里会有一个unit test (for each method), 每个unit test里还会有很多test cases，*assertEquals*("a messege", expected value, actual value)，每个test case都是follow @before @test @after的顺序。  **Inhetitance:** Child can access Parent’s variable and method if and only if they are public or protected. CANNOT access those declared as private or default in the parent class.  Overriding: If you don’t want a method to be overridden by any child, declare it as final.  Shadowing: Child class re-declares a variable that exists in Parent.  **Static binding** happens at compile time, based on type information. （像overloading 一样，调用哪个function是通过parameter type，在compile的时候就知道了的）  **Dynamic binding** happens at runtime time (cannot be sure until running), based on the calling Object. 如果一个subclass的object被存到一个parent class的variable里，调用两个class都有的method会优先调用subclass的（如果subclass有这个method的话），但如果parent class没有这个subclass的method，你就不能对parent class call这个method，除非 cast ，但如果存的不是一个subclass的object就不能cast。random也是一个dynamic binding的例子。  **Abstract class：里面的method可以implement也可以只declare，也可以有variables，cannot be instantiated，subclass of an abstract class can still be abstract.**  **Liskov Substitution Principle**: In other words, methods in the parent class must make total sense for the child class.  **Array of subclass object is not a subclass from an array of parent class object**  **Generic Types：**  drawShapes(ArrayList<Shape> lst)如果是ArrayList<Cirvle>就进不去  drawShapesGeneric(ArrayList<? **extends** Shape> lst)就可以  **UML：**-private, +public, #protected, ~package, static underline  **GIT：**  git branch UserStoryN  git checkout UserStoryN  git add and git commit，To push a new branch to remote: git push --set-upstream origin NEW\_BRANCH  git checkout master （When you believe it’s done）  git pull  git merge UserStoryN  git push to the remote repo |

**Week4 OO Interface, GUI（不是重点，以往的past test从来没考过GUI）**

**Interface**: An interface is a group of **public** methods, declared but **not implemented**,也不能有instance or static variables, except for public static final variables (constants). Implement了一个interface的class可以被当做一个interface的object来看.

**GUI** (Basic Workflow): populate a stage object passed to the start method, a stage has a scene, a scene contains a tree of stuff. Each node of the tree could be one of the following: Layout panes (organize how its subtrees appear), controls (labels, buttons, text fields, etc.), events (e.g. callback methods that defines what happens when key is pressed, mouse is clicked, etc.) class先extends Application（javafx的class，包含了很多implement好的function, 比如start（Stage stage）

|  |  |  |  |
| --- | --- | --- | --- |
| **Grid Pane**  **public** **class** LayoutGrid **extends** Application {  **public** **static** **void** main(String[] args) { *launch*(args);}  **public** **void** start(Stage stage) {  initUI(stage);}  **private** **void** initUI(Stage stage) {  GridPane pane = **new** GridPane();  Scene scene = **new** Scene(pane);  pane.setHgap(10);  pane.setVgap(10);  pane.setPadding(**new** Insets(10));  grid pane不用getChildren,直接add  pane.add(**new** Button("1"), 0, 0);  pane.add(**new** Button("2"), 1, 0);  pane.add(**new** Button("3"), 2, 0);  pane.add(**new** Button("4"), 0, 1);  stage.setTitle("Grid Layout");  stage.setScene(scene);  stage.show();}} | **Flow**  **public** **class** LayoutFlow **extends** Application {  **public** **static** **void** main(String[] args) {*launch*(args);}  **public** **void** start(Stage stage) {  initUI(stage);}  **private** **void** initUI(Stage stage) {  FlowPane pane = **new** FlowPane(5, 10);  Scene scene = **new** Scene(pane, 600, 480);  pane.getChildren().add(**new** Button("North"));  pane.getChildren().add(**new** Button("South"));  \*same for west and east  pane.getChildren().add(**new** TextField("Centre"));  stage.setTitle("Flow Layout");  stage.setScene(scene);  stage.show();}} | Border  **public** **class** LayoutBorder **extends** Application {  **public** **static** **void** main(String[] args) {*launch*(args);}  **public** **void** start(Stage stage) {  initUI(stage);}  **private** **void** initUI(Stage stage) {  BorderPane root = **new** BorderPane();  Scene scene = **new** Scene(root, 60, 48);  Label btop = **new** Label("top");  Label bleft = **new** Label("left");  Label bbottom = **new** Label("bottom");  Label bright = **new** Label("right");  root.setTop(btop);  root.setLeft(bleft);  root.setRight(bright);  root.setBottom(bbottom);  stage.setTitle("Border Layout");  stage.setScene(scene);  stage.show();}} | Complex  **public** **class** LayoutComplex **extends** Application {  //BorderPane里含有一个GridPane  **public** **static** **void** main(String[] args) {*launch*(args);}  **public** **void** start(Stage stage) {initUI(stage);}  **private** **void** initUI(Stage stage) {  BorderPane pane = **new** BorderPane();  pane.setTop(**new** Button("north"));  pane.setBottom(**new** Button("south"));  \*same for west and east  GridPane cpane = **new** GridPane();  **for** (**int** i = 0; i < 9; i++) {  cpane.add(**new** Button("Centre " + i), i % 3, i / 3); //(button #, row #, column #)}  pane.setCenter(cpane);  Scene scene = **new** Scene(pane);  stage.setTitle("Complex Layout");  stage.setScene(scene);  stage.show();}} |

**EventHandler:** A handler is attached to certain events, when the event is detected, the handle method of the handler is invoked. Button.setOnAction(new eventHandler()), eventHandler class **implements** EventHandler<ActionEvent>, 当click on the button，这个class里的handle才就会被调用. 用String msg = ((Button) (event.getSource())).getText()就可以得到button上面的字

四种把eventHanldler和button连起来的方法

|  |  |  |  |
| --- | --- | --- | --- |
| **Alternative 1**  buttonHi.addEventHandler(ActionEvent.***ACTION*** , **new** HiByeEventHandler());  buttonBye.addEventHandler(ActionEvent.***ACTION*** , **new** HiByeEventHandler());  **Alternative 2（inner class）**  不需要再有另一个eventHandler class，直接创建个EventHandler object在里面写handle function  EventHandler<ActionEvent> eventHandler = new EventHandler<ActionEvent>() {             public void handle(ActionEvent e) {      \*how you want the button to react when pressed\* } };           bhi.setOnAction(eventHandler); | **Alternative 3 (anonymous inner class)**  bHi.setOnAction(new EventHandler<ActionEvent>()  {public void handle(ActionEvent event) { \*how you want the button to react when pressed\*}});  **Alternative 4 (Lambda)**  bHi.setOnAction((event) -> {  \*how you want the button to react when pressed\*});  **Alternative 5**  HiByeEventHandler hbh1 = new HiByeEventHandler();  bHi.setOnAction(hbh1); | **KeyMouse extends Application**  **按按键：**scene.setOnKeyTyped(  **new** EventHandler<KeyEvent>() {  **public** **void** handle(KeyEvent event) {  **\*how you want the button to react when pressed\***  System.***out***.println("key pressed: " + event.getCharacter());  **switch** (event.getCharacter()) {}}  **移动鼠标：**scene.setOnMouseClicked  (**new** EventHandler<MouseEvent>() {  **public** **void** handle(MouseEvent e) {   System.***out***.println("mouse clicked: "+ e.getSceneX()+" "+e.getSceneY()); | **Timer**  **public** TimerDemo() {  Timer tickTimer = **new** Timer();  tickTimer.schedule(**new** TickTask(), 1000, 800);  **private** **class** TickTask **extends** TimerTask {  @Override  **public** **void** run() {  System.***out***.println("Tick!!");}}  Timeline timer1 = **new** Timeline(**new** KeyFrame(Duration.*millis*(1500),  **new** TimerHandler("Tick", lbl\_tick)));  timer1.setCycleCount(Animation.***INDEFINITE***);  timer1.play(); |

**Week5 MVC, Observer/Observable, Scrum (重点)**

|  |  |
| --- | --- |
| mvc.jpg  **Model(Observable)**: the internal object, data, application state  **View(Observer)**: the user interface, reflecting the changes in model.  **Controller:**  1. receive an event triggered from the view.  2.can manipulate the model and change the application state.  3. connects the model and the view, so that when change to the model happens, the model and notify the view to make the corresponding change. | **Observer/Observable (java自带的)**  **extends Observable:**  addObserver(): add an observer  setChanged(): set the “changed” flag to be true  notifyObservers(): tell all observers about the change  后两个method一般一起用，当model里的method让object的状态产生变化时就notifyObservers，但如果没有变化就不要call这两个method. Model里的notifyObserver对应的就是observer里的update。  **implements Observer:** update(Observable o, Object arg) which is called when the observer is notified about a change by the observable. Model里的change你想要怎么样在view里展现出来都写在Update里.  **Summary of MVC:**  Better extensibility and reusability  Supports better collaboration  **Scrum**  Waterfall V.S. Agile (preferred, Iterative approach)  Scrum is an agile methodology |

**Week6 Design Patterns: singleton, iterator (重点)**

|  |  |
| --- | --- |
| **Observer/Observable (User Defined)**  When certain objects need to be informed about the changes occurred in other objects.example: MVC GUI  **Advantage**：It supports the principle of loose coupling between objects that interact with each other. It allows sending data to other objects effectively without any change in the Subject or **Observer** classes. **Observers** can be added/removed at any point in time.  **Implementation:**  **public** **interface** Observer {**public** **void** update ();}  **public** **class** Observable {  **private** ArrayList<Observer> observers = **new** ArrayList<Observer>();  **public** **void** attach(Observer o) {**this**.observers.add(o);}  **public** **void** detach(Observer o) {**this**.observers.remove(o);}  **public** **void** notifyObservers() {**for** (Observer o : **this**.observers) {o.update();}}}  note:如果没有变化就不要call这两个notifyObservers()  **Advanced Issues**: Push and Pull communication methods  **push model**: observable一次性把所有一大堆信息全部push给所有observer，有用的没用的全都有，这些信息observer可以自己决定用不用。有点浪费的是他涵盖了很多没用的信息，有可能很占空间。  **pull model**:每次有新的change时observable只call notify这个method，告诉observer something has changed，但不告诉observer什么东西change了，observer要是想知道发生了什么需要自己去observable那里pull新的信息，缺点就是会有很多threads同时访问observable，需要排队解决什么的，所以运行时间可能会变慢。（concurrency issue, b/c multiple access on Observable.）  对于这两种方法而言没有说哪个更好哪个不好，都有自己的pros and cons，用哪个取决于你的application，你的program，你的需求是什么 | **Iterator**  **Advantage**: have a unified mechanism to traverse any collection， hide the internal implementation of the collection, i.e., how the elements are really stored  **Implementation:**  **The Collection class（其实还是用ArrayList在存东西）** implements the Iterable< collection class > interface, and the iterator() method, which returns an iterator that points to the beginning of the collection.(相当于加上了index)  例如：**public** **class** SongCollection **implements** Iterable<Song>  **public** Iterator<Song> iterator() {**return** **new** SongCollectionIterator(songs);}  **The Collection’s iterator class** implements the Iterator<collection class> interface, includes:  1.constructor: create a new iterator pointing at the beginning of the collection  2.hasNext(): return False iff the iterator is at the end of the collection  3.next(): return the current item, move iterator one step forward.  **public** **class** SongCollectionIterator **implements** Iterator<Song> {  **private** ArrayList<Song> songs;  **private** **int** indexKey;  **public** SongCollectionIterator(ArrayList<Song> s) {**this**.songs = s; indexKey = 0；}  **public** **boolean** hasNext() {**return** **this**.indexKey < **this**.songs.size();}  **public** Song next() {Song r = **this**.songs.get(indexKey); indexKey++; **return** r;}  Collection c;  Iterator it = c.iterator();  while (it.hasNext()) { print(it.next()); }  另一种java自带的接口:  for (Object o: Collection c) { print(o); } |

**Week7 Design Patterns: Strategy, Command, Composite (重点)**

|  |  |  |
| --- | --- | --- |
| **Strategy**  Use when you have a family of algorithms, and you want to use them interchangeably. You want to be able to change the algorithm being used dynamically at runtime. You want to encapsulate the algorithms.  **Advantage**: Separate algorithms into classes that can be plugged in at runtime, Strategy enables the clients to choose the required algorithm, without using a "switch" statement or a series of "if-else" statements.  **Implementation:**  1.Create a common Interface for all the algorithms/strategies in the family.  2.Implement the interface for each concrete strategy.  3.The class using the strategies has the strategy Interface object去存我们的concrete strategy, which can be set using a setter method.  4.Clients use the setter method to change strategies dynamically  **public** **interface** RobotStrategy {**public** String nextCommand();}  **public** **class** RobotStrategyDefensive **implements** RobotStrategy {  **public** String nextCommand() {**return** "defense";}}  **public** **class** Robot { **private** String name; **private** RobotStrategy strategy;  **public** Robot(String name) { **this**.name = name; **this**.strategy = **new** RobotStrategyNormal();}  **void** move() { System.***out***.print(name + " makes a move: ");  String command = **this**.strategy.nextCommand();  System.***out***.println(command);} robot.setStrategy(**new** RobotStrategyAggressive()); **for** (**int** i = 0; i < 10; ++i) {robot.move(); // these moves are aggressive  **public** **interface** CompareStrategy {  **public** **static** **final** **int** ***LESS*** = -1; ***EQUAL*** = 0; ***GREATER*** = 1;  **public** **int** compare(String s1, String s2);}  **public** **class** CompareStrategyNormal **implements** CompareStrategy {  **public** **int** compare(String s1, String s2) {  **if** (s1.compareTo(s2) < 0) {**return** CompareStrategy.***LESS***;}  **else** **if** (s1.compareTo(s2) > 0) {**return** CompareStrategy.***GREATER***;}  **return** CompareStrategy.***EQUAL***;}}  **public** **class** OrderedStringList {  **private** ArrayList<String> list = **new** ArrayList<String>();  **private** CompareStrategy strategy;**public** OrderedStringList(CompareStrategy strategy) {**this**.strategy = strategy;}  **public** **void** add(String s) {**for** (**int** i = 0; i < **this**.list.size(); i++) {  **if** (strategy.compare(s, **this**.list.get(i)) == CompareStrategy.***LESS***) { **this**.list.add(i, s);**return**;}}**this**.list.add(s);} | **Command**  You want to send requests/commands to a receiver object, to make the receiver object perform various actions. a hard drive could queue up a sequence of write command, **reorder them** to **optimize performance,** then **execute the commands in batch.**  **Advantage：**It decouples the classes that invoke the operation from the object that knows how to execute the operation. It allows you to create a sequence of **commands** by providing a queue system. Extensions to add a new command is easy and can be done without changing the existing code.  **Implementation:**  1.Create a common interface for all commands for a given receiver object which include an execute() method. And keeps a reference of the receiver object.  **public** **interface** BalloonCommand {**public** **abstract** **void** execute();}  2.Implement the interface for each concrete command (implement the execute() method). Uses the action methods of the receiver class.  **public** **class** InflateCommand **implements** BalloonCommand {  **private** Balloon balloon; **private** **int** amount = 0;  **public** InflateCommand(Balloon balloon, **int** amount) {  **this**.balloon = balloon; **this**.amount = amount;}  **public** **void** execute() {**this**.balloon.inflate(amount);}}  3.Client: instantiate a receiver object, instantiate concrete commands. Issue the command by: command.execute()  Balloon b1 = **new** Balloon("RED", 100);  BalloonOperator operator = **new** BalloonOperator();//the invoker  operator.acceptCommand(**new** InflateCommand(b1, 20));  operator.operateAll();  4.Invoker: stores the commands, and issue them by calling execute() on them.  **public** **class** BalloonOperator {  ArrayList<BalloonCommand> commandQueue;  **public** BalloonOperator() {  commandQueue = **new** ArrayList<BalloonCommand>();}  **public** **void** acceptCommand(BalloonCommand command) {  **this**.commandQueue.add(command);}  **void** operateAll() {  **for** (BalloonCommand command: **this**.commandQueue) {  command.execute();}commandQueue.clear(); } | **Composite**  When working with an object with recursive structure. a component of class A is composed of one or more components of class A. (composed graphics and arithmetic expressions)  **Advantage**: easy to use for the client, **no if statement is needed** in order to handle Composite and Simple differently.  **Implementation:**  **1.**Define a common interface for the simple component and the composite component.  **public** **interface** GraphicComponent {**public** **void** paint();}  2. Implementations these interface, one for simple component, one for the composite implementations.  **public** **class** GraphicSimple **implements** GraphicComponent {  **private** String name = "";  **public** GraphicSimple(String name) {**this**.name = name;}  **public** **void** paint() {System.***out***.println(name + ": simple component");  3.The composite component has an addElement() method which adds a component (simple or composite) to the composite.  **public** **class** GraphicComposite **implements** GraphicComponent {  **private** ArrayList<GraphicComponent> children = **new** ArrayList<GraphicComponent>(); **private** String name = "";  **public** GraphicComposite(String name) {**this**.name = name;}**public** **void** paint() {System.***out***.println(**this**.name + ": composite component");**for** (GraphicComponent c: **this**.children) {c.paint();}}  **public** **void** add(GraphicComponent c) {**this**.children.add(c);}}  4.The client uses the methods defined in the interface.  GraphicSimple carBody = **new** GraphicSimple("rectangle");  GraphicSimple wheel1 = **new** GraphicSimple("circle");  GraphicSimple treeTop = **new** GraphicSimple("triangle");  GraphicSimple treeBotm = **new** GraphicSimple("rectangle");  GraphicComposite car = **new** GraphicComposite(“group 1");  GraphicComposite tree = **new** GraphicComposite("group 2");  car.add(carBody); car.add(wheel1); car.add(wheel2);  tree.add(treeTop); tree.add(treeBottom);  GraphicComposite wholePic = **new** GraphicComposite("main group"); wholePicture.add(car); wholePicture.add(tree);wholePicture.paint(); |

**Week8 Design Patterns: Factory, Builder(重点)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Factory（**simple creation process**）**  **Advantage**：Creates objects without exposing the instantiation logic to the client.  The product of a factory could also be a strategy, a command等  **Implementation: factory是不能有parameters的**  1.Create a base class or interface for the product (Food)  2.Implement concrete product classes by extending the base class (Burger, Pizza, Salad, etc. )  3.Create the Factory class with a createProduct(String productID) method, returns objects of different types according to productID  **public** **class** Food {}  **public** **class** Pizza **extends** Food {}  **public** **class** Salad **extends** Food {}  **public** **class** FoodFactory {  **public** Food createProduct(String product) {  **if** (product.equals("Burger")) **return** **new** Burger();  **if** (product.equals("Fries")) **return** **new** Fries();  **return** **null**;} | **Singleton**  Application needs **only one** instance of an object, e.g., logger object. Also, provides a global point of access to that instance.  **Advantage：**Instance control: Singleton prevents other objects from instantiating their own copies of the Singleton object, ensuring that all objects access the single instance.  **Implementation:**  1. object created when the program starts  Public final class Singleton{  Private static final Singleton INSTANCE = new Singleton();  Private Singleton(){}  Public static Singleton getInstance(){return Instance}}  2. object created when the first time you use the Singleton class  Public class Singleton{  Private static Singleton instance = new null;  Public static synchronized Singleton getInstance(){  If (instance ==null)instance = new Singleton(); return instance;}  要创建singleton object只能通过Singleton s1 = Singleton.*getInstance*(); 因为singleton的constructor是private的 | | **Builder（complex creation process）**  **Advantage**：able to customize many attributes of the object, and separates object construction from its representation. The client doesn’t need to see the constructor of the product class  **Implementation (Basic)**  **1.**Define the Product class with different attributes (to be customized) and their setter methods.  2.Define a Builder class that keeps the options for setting the Product's attributes, and has methods (buildParts()) for building different parts of the product.  3. Builder has a getProduct() method that create a Product object, configures its attributes, and returns the Product object.  **public** **class** Pizza {**private** String name; **private** **boolean** extraCheese, extraSauce;  **public** Pizza(String name) {**this**.name = name; **this**.extraCheese = **false**; **this**.extraSauce = **false**;}  **public** **void** setExtraCheese(**boolean** extraCheese) { **this**.extraCheese = extraCheese;}  **public** **void** setExtraSauce(**boolean** extraSauce) {**this**.extraSauce = extraSauce;}  **public** **class** PizzaBuilder {  **private** String name; **private** **boolean** extraSauce = **false**, extraCheese = **false**;  ;**public** PizzaBuilder(String name) {**this**.name = name;}  **public** **void** addExtraSauce() {**this**.extraSauce = **true**;}  **public** **void** addExtraCheese() {**this**.extraCheese = **true**;}  **public** Pizza getPizza() {Pizza p = **new** Pizza(**this**.name);  p.setExtraCheese(extraCheese); p.setExtraSauce(extraSauce); **return** p;} | |
| **Builder (Concrete Builder)**  Extends from the Builder and make a concrete builder with a specific configuration.  **public** **class** HawaiianPizzaBuilder **extends** PizzaBuilder {  **public** HawaiianPizzaBuilder() {  **super**("Hawaiian");  **this**.addPineapple();  **this**.addPepperoni();}}  PizzaBuilder hawaiianBuilder = **new** HawaiianPizzaBuilder();  Pizza h0 = hawaiianBuilder.getPizza();  System.***out***.println(h0); | | **Builder(Director-like cashier)**  **public** **class** PizzaDirector { **private** PizzaBuilder builder;  **private** ArrayList<Pizza> pizzas = **new** ArrayList<Pizza>();  **public** PizzaDirector() {}  **public** **void** construct() { builder = **new** HawaiianPizzaBuilder();  builder.addExtraCheese(); pizzas.add(builder.getPizza()); pizzas.add(builder.getPizza()); builder = **new** DeluxePizzaBuilder();  pizzas.add(builder.getPizza()); pizzas.add(builder.getPizza());  pizzas.add(builder.getPizza());}  **public** ArrayList<Pizza> getPizzas() {**return** **this**.pizzas;}}  // use the director to construct a bunch of pizza  PizzaDirector director = **new** PizzaDirector();  director.construct();  ArrayList<Pizza> pizzas = director.getPizzas();  **for** (Pizza p : pizzas) {System.***out***.println(p);} | | **Builder (Chain Builder-like restaurant crew)**  **public** **class** PizzaChainBuilder {  **private** String name; **private** **boolean** extraSauce = **false**;  **private** **boolean** extraCheese = **false**;  **public** PizzaChainBuilder(String name) {**this**.name = name;}  //就可以一直点一直点一直加那些method，之前这些method return void的时候就不可以把这些method全写一行,其他都和普通builder是一样的  **// return the builder itself rather than void**  **public** PizzaChainBuilder addExtraSauce() {  **this**.extraSauce = **true**; **return** **this**;}  **public** PizzaChainBuilder addExtraCheese() {**this**.extraCheese = **true**;  **return** **this**;}  Pizza p1 = **new** PizzaChainBuilder("TheLarry")  .addExtraCheese().addExtraSauce().addPepperoni().getPizza();  System.***out***.println(p1); |

**Week9 JavaIO, Regular expressions (重点)**

|  |  |  |
| --- | --- | --- |
| **Byte Streams (**unbuffered**)**: handle I/O of raw binary data, Reads and writes one byte at a time.  EX. FileInputStream in = new FileInputStream(“input.txt”)  FileOutputStream in = new FileOutputStream(“output.txt”)  Int c;  While ((c = in.read())!=-1){out.write(c);}  **Character Streams(**unbuffered **)**: handle I/O of character data, automatically handling translation to and from the local character set. Reads and writes one char (two bytes) at a time.  EX. FileReader in = new FileReader(“input.txt”)  FileWriter in = new FileWriter(“output.txt”)  Int c;  While ((c = in.read())!=-1){out.write(c);}  //这个function允许你input进去string，你在console写“nishisheiya”会把每个char放进array里, 这是直接**在console里read characters**  **char**[] c = **new** **char**[10];  **try** { **for** (**int** i = 0; i < c.length; i++) {  c[i] = (**char**) System.***in***.read();}}// System.in is an InputStream  **catch** (IOException e) {System.***out***.println(e);} | **Scanner:** allows a program to read and write formatted text.  Scanner scan = **new** Scanner(System.***in***);//先创建一个scanner我们才可以得到user input  System.***out***.println("How many balloons?");  **int** numBalloons = Integer.*parseInt*(scan.nextLine());  System.***out***.println("What colour?");  String colour = scan.nextLine();  **这是从file读取**  **try** {BufferedReader in = **new** BufferedReader(**new** FileReader("words.txt"));  Scanner s = **new** Scanner(in);  **while** (s.hasNextLine()) { String line = s.nextLine();  **if** (line.startsWith("ab")) {System.***out***.println(line);}  s.close();//这里是scanner close  } **catch** (FileNotFoundException e) {  System.***out***.println(e);}  **File 同时read和write的例子**  **public** **static** **void** fileReadWrite() **throws** IOException {//这里thro一个exception之后就不会报错  这个function就是把words.txt里的文件写用out.write(c)写到"words-copy.txt"去了，同时把o字母替换成\*\*\*  FileReader in = **null**; FileWriter out = **null**;  **try** {in = **new** FileReader("words.txt");  out = **new** FileWriter("words-copy.txt");//在这里就创建了个新的file  **int** c; System.***out***.println("Copying...");  **while** ((c = in.read()) != -1) {//read（）如果没有更多东西了就会return-1  **if** (c == 'o') {//把原文件里所有o用\*\*\*替换掉  out.write("\*\*\*");}**else** {out.write(c);}} System.***out***.println("Done!");  } **finally** {//和我们之前学的try catch一样，finally是无论如何都会run的  **if** (in != **null**) {in.close();}**if** (out != **null**) {out.close();}} | **Buffered Streams**: optimize input and output by reducing the number of calls to the native API.  Advantage: for read: one disk access reads/writes (batch write) a batch of data from/to the disk to a memory area (call buffer), then Java read() gets data from the buffer. Much smaller number of disk access, much more efficient. **The System.in and System.out are essentially files.**  EX. BufferedReader in = new BufferedReader (FileReader(“input.txt”))  **这也是从console直接读取**，read a line at a time  BufferedReader lineInput = **new** BufferedReader(**new**InputStreamReader(System.***in***));  String line;  **try** {**while** ((line = lineInput.readLine()) != **null**) {  System.***out***.println("line = " + line + ", size=" + line.length());}} **catch** (IOException e) {  System.***out***.println(e);}  // Alternatively, use a scanner  Scanner sc = new Scanner(System.in);  **Read from files,** read words and count the number of lines in the file, 把words.txt里的q开头的所有单词都print出来了，和q开头单词的总数  **try** {FileReader fr = **new** FileReader("words.txt");  BufferedReader lineInput = **new** BufferedReader(fr);  String line; **int** count = 0;  **while** ((line = lineInput.readLine()) != **null**) {  **if** (line.startsWith("q")) {System.***out***.println(line);count++;}}  fr.close();//这里就就把那个file关上了  System.***out***.println(count);} **catch** (FileNotFoundException e) {System.***out***.println(e);} **catch** (IOException e1) {System.***out***.println(e1);} |

**Regular Expression (string matching)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Pattern** | **Matches** | **Explanation** | **Pattern** | **Explanation** |
| **a\*** | **“” “a” “aa”** | **Zero or more times** | **\t** | **A tab** |
| **b+** | **“b” “bb”** | **One or more times** | **\n** | **A new line** |
| **ab?c** | **“ac” “abc”** | **Zero or one time** | **.** | **Any character** |
| **[abc]** | **“a” “b” “c”** | **One from a set** | **\d** | **A digit [0-9]** |
| **[a-c]** | **“a” “b” “c”** | **One from a range** | **\D** | **A non-digit [^0-9]** |
| **[abc]\*** | **“” “acbccb”** | **combination** | **\s** | **A whitespace[\t\n\x0B\f\r]** |
| **^** | anchors | **Matches beginning of the line如果没有会自动加** | **\S** | **A non-whitespace [^\s]** |
| **$** | anchors | **Matches the end of the line如果没有会自动加** | **\w** | **A word char [a-zA-Z\_0-9]** |
| **\** | **\^, \$, \\*** | **Escape, matches the actual symbol** | **\W** | **A non-word char [^\w]** |
| **[^abc]** | **negation** | **Any char except a,b or c** | **[a-z&&[def]]** | **Intersection, d, e, or f** |
| **[a-zA-Z]** | **range** | **a thru z or A thru Z inclusive** | **[a-z&&[^bc]]** | **Subtraction, a thru z except for b and c** |
| **[a-d[m-p]]** | **union** | **A thru d or m thru p** | **[a-z&&[^m-p]]** | **Subtraction ,a thru z but not m thru p** |
| **X{n}** | **Exactly n times** | **X{n,} at least n times** | **X{n,m}** | **At least n but no more than m times** |
| (a|z) | **“OR”, matches a or z** |  |  |  |

EX. to match the student number (\d{9}\d?) or (\d{9,10}), to match the grade, ((100)| ([1-9]?\d))一百，两位数，一位数的情况分别考虑, 并且不能出现09这样的分数，所以第一位只能[1-9]

Note: 在java里要写"([\\d\\d\\d)ABC\\1](file://d//d//d)ABC//1)" 才行，两个slash

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Capturing Groups**  Capturing groups allow you **to treat multiple characters** as a single unit. Use括号to group. For example, (BC)\* means zero or more instances of BC, e.g., BC, BCBC, BCBCBC, etc.  1.groups are numbered by counting their opening parentheses from left to right. Group0永远是最大的那个整体  2.**Backreference**  The section of the input string matching the capturing group(s) is saved in memory for later recall via backreference.  A backreference is specified in the regular expression as a backslash (\) followed by a digit indicating the number of the group to be recalled.   |  |  | | --- | --- | | pattern | Example matching string | | (\d\d)\1 | 1212 | | (\w\*)\s\1 | asdf asdf |   \*用matcher.groupCount()可以得到group总数，用m.group(i)可以得到第1个group | **Regex in Java**  A proper regex for this set matches all strings in this set and does NOT match any string NOT in this set.  Pattern pNaturalNum =Pattern.*compile*("(0|[1-9]+\\d\*)");（a DFA if built here by the compile）  Matcher m= pNaturalNum.matcher(string);  System.out.println(m.matches()); (match() will return true or false)  Pattern pCircle=Pattern.*compile*("^Circle$");(也可以match一模一样的string)  **简易版本**：System.out.println ( Pattern.matches(“a\*b”, “aaaaab”));  **match() V.S. find()**  match()一定要是一模一样的match了才会return true  find（）只要有一个sub-string match了就会return true  p = Pattern.*compile*("(\\d\\d\\d)ABC\\1");//ABC后面一定要跟group1里的内容，which is 三个digits  m = p.matcher("123ABC123"); System.***out***.println(m.matches());//true  m = p.matcher("123ABC456"); System.***out***.println(m.matches());//false |

**Week10 Finite State Machine / DFA (重点，应该会有一道大题，还是画DFA的图，有trapping state（到这个state后直接就break ）**

|  |  |
| --- | --- |
| **分别用DFA和regex来判断一个string是否是5个倍数**  **public** **static** **boolean** recognise5Regex(String s) {  Pattern p = Pattern.*compile*("^\\d\*(0|5)$");  Matcher m = p.matcher(s);  **return** m.matches();}  **public** **static** **boolean** recognise5FSM(String s) {  **char**[] c = s.toCharArray(); // so you can get a char by c[i]  **int** len = s.length(); **int** i = 0;  **int** state = 0; // Start out in the initial state  **while** (i < len) {  **switch** (state) {  **case** 0://注意他这个每个state，也就是0，1都有3条通向外面的path！！  **if** (c[i] == '0' || c[i] == '5') state = 1;//accepting state  **else** **if** ('0' <= c[i] && c[i] <= '9') state = 0;  **else** state = 2;//trapping state **break**;  **case** 1:  **if** (c[i] == '0' || c[i] == '5') state = 1;  **else** **if** ('0' <= c[i] && c[i] <= '9') state = 0;  **else** state = 2; **break**;  **case** 2://trapping state **break**;} i = i + 1;}  **return** state == 1;//看当前state是否是accepting state来决定accept与否} | **public** **void** parseMarks2() **throws** IOException {  BufferedReader inputStream = **null**;//读取一行  **try** {  Pattern pColons = Pattern.*compile*("^:{14}$");//::::::::::::::这一串符号  Pattern pStartMarksLine = Pattern.*compile*("MARKS For Assignment 1, Part 2");  Pattern pGUIMarks = Pattern.*compile*("^GUI:\\s\*(\\d(\\.\\d)?)/5\\s\*$");//这里的\\s\*是空格,分数是\d/5，几点几不一定要存在  Pattern pCodeMarks = Pattern.*compile*("^CODE:\\s\*(\\d(\\.\\d)?)/5\\s\*$");//一模一样的对照这个string  Pattern pEndMarksLine = Pattern.*compile*("^END MARKS$");//(.\*)是utorid  Pattern pUtorid = Pattern.*compile*("^(.\*)/JugPuzzleGame/src/JugPuzzleGUIController\\.java$");  inputStream = **new** BufferedReader(**new** FileReader(basePath + "all.txt"));//在所有directory里找到all.txt这个file  **int** state = 0;// State 0 is before ":::::::::::::::::"，一共7个state，每行一个state，等遇到新的“:::::::::::”就回到state0  Matcher m; String l, utorid = ""; **float** guiMark = 0, codeMark = 0; lineNumber = 0;  **while** ((l = inputStream.readLine()) != **null**) { lineNumber++;  **switch** (state) {  **case** 0: // state before :::::::  m = pColons.matcher(l);  **if** (m.matches()) {//如果是:::::::了，initialize所有新的信息，因为已近读到了一个新的student  utorid = ""; guiMark = 0; codeMark = 0; state = 1;} **break**;  **case** 1: // after reading the opening ::::::: m = pUtorid.matcher(l);  **if** (m.matches()) {//如果读到utorid那一行了  utorid = m.group(1);//用group1把这个学生的utorid记下  state = 2;//继续往下读} **else** {//如果出错了，出一个提示语，但不真的报错，结束程序 error("Expecting utorid line");**return**;}**break**;  **case** 2: // after reading the utorid m = pColons.matcher(l);  **if** (m.matches()) state = 3; **else** { error("Expecting colons"); **return**;}**break**;  **case** 3: // after reading the :::: below the utorid line  m = pStartMarksLine.matcher(l); **if** (m.matches()) {state = 4; **break**;}//如果不match“MARKS For Assignment 1, Part 2”  m = pColons.matcher(l);//但match“:::::::”的话 **if** (m.matches()) { error("Expecting start marks line");**return**;}**break**;  **case** 6: // after reading the CODE mark  m = pEndMarksLine.matcher(l);//读到“END MARKS”  **if** (m.matches()) {//因为我们再main里call的method，我们parse时utoridToStudent里应该已经是有所有student的object了，我们就可以通过当前读到的学生的utorid的到treeMap里的student本人，在把它们的marks赋值给它们各自  **this**.utoridToStudent.get(utorid).setGuiMark(guiMark); **this**.utoridToStudent.get(utorid).setCodeMark(codeMark);  state = 0;//当前学生的信息就收集齐了，回到state0准备读取新的学生的数据！}}}  // Checks at the end of reading，因为如果我们读到了'END MARKS'，state一定是等于0的  **if** (state != 0) {error("Expected end of file");} **else** {**for** (String s : **this**.utoridToStudent.keySet()) {  System.***out***.println(utoridToStudent.get(s));}}} **finally** {**if** (inputStream != **null**) {inputStream.close();}}} |

**Week11 Floating Point (重点) \*不能用小数做loop counter，会变成infinite loop**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Decimal-i**  **Binary: -i**  **IEEE-754 Floating Point Format**  sign s (1 bit) | exponent e (8 bits) | mantissa M(23 bits)  (-1)s \* (1+M)\*2e-127  \*range of exponent is from  (0-127) to (255-127) = -127 to 128  \*mantissa第一位永远是1，就不存他了  \*sign：1 for negative and 0 for positive  **Convert binary to decimal**  Integer: 1100 1111  1(27)+1(26)+0(25)+0(24)+1(23)+1(22)+1(21)+1(20)=207  Faction:0.0111  0(2-1)+1(2-2)+1(2-3)+1(2-4)=0.25+0.125+0.0626=0.4375  \*在binary里mantissa一定是小数了，因为他的整数位一定是1并且被省略了，要用小数的方法求出后加1  \*exponent就用整数的方法求（**exponent要减127**）  \*最后写成(-1)s \* (1+M)\*2e-127的形式  **Convert decimal to binary**  \*若是整数位不是0，就一直除2，一直除到整数位是1，然后exponent就是你除的2的次数。若整数位是0，就一直乘2，一直乘到整数位是1，然后乘上2-int（int就是你乘的2的次数，记住是负的），再分别把sign，mantissa，exponent变成binary的形式（**exponent要加127**后再变）  \*mantissa（除去1的小数位）乘2，用得出的结果的小数位再乘2，再用上一位的小数位乘2，保证每次乘2 的数的整数位是0，记住每次乘2后的结果的整数位（会是0或1），一直乘到有23位bit就行，按顺序读（可能还需要再往后乘几位 for rounding purpose）  \*convert int to binary:一直用小学方法除2，每次的remainder就是bit，下一次用quotient继续除2，一直除到quotient为0为止（**然后从下往上读**）  \*若是要convert negative int to binary，可以找出他的positive binary，然后用1s complement的方法，把0，1互换。或者前面加一个sign bit | **special values**  Zero: 0[00000000]00000000000000000000000（“^0{32}$”）  Positive Infinity: 0[11111111]00000000000000000000000（"^01{8}0{23}$”）  Negative Infinity：1[11111111]00000000000000000000000（"^11{8}0{23}$”）  Not a Number: \*[11111111]-anything-but-all-zero-（"^[01]1{8}[01]{23}$"）  Zero To 255: "([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])"  Natural Number: "(0|[1-9]+\\d\*)"  **Overflow**：overflow is the largest representable number  0[11111110]11111111111111111111111 = +1.11111111111111111111111(binary) x 2^(127) \*exponent不能全是1不然就会变成not a number  **Underflow**：Underflow is the smallest positive representable number  **(not really)**0[00000001]00000000000000000000000 =1.00000000000000000000000(binary) x 2^(-126) ) \*exponent不能全是0不然就会变成0  (**real underflow**)0[00000000]00000000000000000000001=0.00000000000000000000001(binary) x 2^(-126) = 2^(-23) x 2^(-126) = 1 x 2^(-149)  **Denormalized Numbers**:  我们上面是只允许在整数位是1,用负的exponent来达到最小数，但现在可以在exponent变小的同时也把mantissa变小，不在是一点几，而是0.00…1  0[00000001]00000000000000000000000=+1.00000000000000000000000x2^(-126)=1.17549435E-38 0[00000000]10000000000000000000000=+0.10000000000000000000000x2^(-126)=5.877472E-39 0[00000000]01000000000000000000000=+0.01000000000000000000000x2^(-126)=2.938736E-39   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 210 | 211 | 212 | 213 | 214 | 215 | | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 | 2048 | 4096 | 8192 | 16384 | 32768 | | 0000  0001 | 0000 0010 | 0000 0100 | 0000 1000 | 0001 0000 | 0010 0000 | 0100 0000 | 1000 1000 |  |  |  |  |  |  |  |  |   **Rounding：**  **1.round to the nearest even number (17.5 to 18, 16.5 to 16)**  **2.at the 23rd bit, we must round to the nearest even, 看第23后3位来决定**  a. If the next (24th) bit is a 0, then you round down directly (do nothing)  b. If the next bit is a 1, followed by either a 10, 01, or 11, you round up (add 1 to the mantissa's 0 least significant digit.)  c. If the next three digits are "100" this is a tie (we are midway (.5) between two representable numbers). In this case:  i. If the last number in the mantissa (23rd bit) is a 1, then round up  ii. If the last number in the mantissa (23rd bit) is a 0, then round down (do nothing)  (i.e. if the mantissa is odd, we're adding 1, if it's even, do nothing. Hence, this is considered rounding to even.)  ex. 001 100🡪010, 110 011🡪110  **Machine Epsilon**: eps is such that 1 + eps is the smallest possible mantissa you can get that is > 1. Machine Epsilon is the best precision you can have in the mantissa.再小了machine就比不了了  For single precision, eps = 1 x 2^{-23} ≈ 1.19e-7 (i.e., if you add 1.0 by 1e-7, nothing’s gonna change.)  For double precision, eps = 1 x 2^{-52} ≈ 2.22e-16  \*非常大的数和非常小的数相加一定要先把小的数全部加起来，**可以先sort**，从小的开始加。And adding a very small quantity to a very large quantity can mean the smaller quantity falls off the end of the mantissa. But if we add small quantities to each other, this doesn’t happen. And if they accumulate into a larger quantity, they may not be lost when we finally add the big quantity in.  \*Avoid checking equality between two numbers using “==”  don’t check this condition: x == 0.207 ○ check this: (x >= 0.207-0.0001) && (x <= 0.207+0.0001) ○ or check this: abs(x - 0.207) <= 0.0001  \*当fraction转换成binary（with infinite digit）会被rounded，convert back to decimal的时候就会不准确，但只有小数点后7位是significant的，我们只保留这7位就够了 |