|  |
| --- |
| **Exercise 6.2**  The documentation contains the following sections:   * Overall description of the class: its purpose and how to use it * A brief summary of the fields * A brief summary of the constructors * A brief summary of the methods * A detailed description of the fields * A detailed description of the constructors * A detailed description of the methods |
| **Exercise 6.3**  **Both check whether the string contains their first argument. The single-argument version of the method starts the match at index 0 within the string. The two-argument version starts the match at the position of the second argument. Both return true if there is a match and false otherwise.** |
| **Exercise 6.4**  Yes:  public boolean endsWith(String suffix) |
| **Exercise 6.5**  Yes: public int length() |
| **Exercise 6.7**  Signature: public String trim() Example: String trimmedText = text.trim(); |
| **Exercise 6.8**  add the line:  input = input.trim();  to the start() method of SupportSystem |
| **Exercise 6.9**  add the line:  input = input.toLowerCase();  to the start() method of SupportSystem |
| **Exercise 6.10**  boolean |
| **Exercise 6.11**  if(input.equals("bye")) {  finished = true; } |
| **Exercise 6.12**  Package: java.util It generates random numbers  An instance is created by using one of the two constructors: Random random = new Random(); Random random = new Random(seed);  To generate a random integer: random.nextInt(); |
| **Exercise 6.13**  Random random = new Random(); random.nextInt(); |
| **Exercise 6.14**  import java.util.Random;  public class RandomTester  {  private Random random; // the random number generator  public RandomTester()  {  random = new Random();  }  public void printOneRandom()  {  System.out.println(random.nextInt());  }  public void printMultiRandom(int howMany)  {  for(int i = 0; i < howMany; i++) {  printOneRandom();  }  }  } |
| **Exercise 6.15**  0,1,2,...,99 |
| **Exercise 6.16**  public int throwDie()  {  return random.nextInt(6) + 1;  } |
| **Exercise 6.17**  public String getResponse()  {  int answer = random.nextInt(3);  if(answer == 0) {  return "yes";  }  else if(answer == 1) {  return "no";  }  else {  return "maybe";  }  }  A switch statement could also be used:  switch(random.nextInt(3)) {  case 0: return "yes";  case 1: return "no";  case 2: return "maybe";  }  A more sophisticated alternative (using arrays from chapter 7) would be:  public String getResponse()  {  String[] responses = {  "yes",  "no",  "maybe",  };  return responses[random.nextint(responses.length)];  } |
| **Exercise 6.18**  private ArrayList<String> responses;  public RandomTester() // constructor  {  responses = new ArrayList<>();  responses.add("yes");  responses.add("don't know");  ...  }  public String getResponse()  {  return responses.get(random.nextInt(responses.size()));  } |
| **Exercise 6.19**  public int getOneRandom(int max)   {  return random.nextInt(max) + 1;  } |
| **Exercise 6.20**  public int getOneRandom(int min, int max)   {  return random.nextInt(max - min + 1) + min;   }  public int getOneRandom(int max)   {  return getOneRandom(1, max);  } |
| **Exercise 6.23**  When you add more responses these also get shown randomly together with the existing ones. This is because the length of the list with responses is used when generating the random number. |
| **Exercise 6.25**  Methods in HashMap that depend on the type parameters:   * Set<Map.Entry<K,V>> entrySet() * V get(Object key) * Set<K> keySet() * V put(K key, V value) * void putAll(Map<? extends K,? extends V> m) * V remove(Object key) * Collection<V> values()   Yes, the same type could be used for both parameters. |
| **Exercise 6.26**  Call its size() method. |
| **Exercise 6.27**  public class MapTester {  private HashMap<String, String> phoneBook = new HashMap<> ();  public MapTester()  {  contacts.put("Charles Nguyen", "(531) 9392 4587");  contacts.put("Lisa Jones", "(402) 4536 4674");  contacts.put("Hiroshi Ito", "(998) 5488 0123");  }  public void enterNumber(String name, String number)  {  phoneBook.put(name, number);  }  public String lookupNumber(String name)  {  return phoneBook.get(name);  }  } |
| **Exercise 6.28**  It overwrites the previous value associated with the key. |
| **Exercise 6.29**  Both values stay in the map. HashMaps only uses the key to distinguish entries - not the values. |
| **Exercise 6.30**  phoneBook.containsKey("Homer Jay Simpson"); |
| **Exercise 6.31**  It returns null. |
| **Exercise 6.32**  **Call the** keySet **method to return the** Set **of keys, and then iterate over the set.**  for(String name : phoneBook.keySet()) {  System.out.println(name); } |
| **Exercise 6.34**  Similarities between HashSet and ArrayList Both store an arbitrary number of objects It is possible to add objects (with the add method) It is possible to remove objects (with the remove method) Both have a size() Both have provide an iterator() method to go through all the elements  Differences: In a HashSet each object can only appear once in the set (because it is a Set). In a ArrayList an Object can appear multiple times. An ArrayList is ordered while a HashSet is not ordered. |
| **Exercise 6.35**  Using HashSet guarantees that there are no duplicate elements, but it does not keep the order of the words. |
| **Exercise 6.36**  It creates empty strings - which was probably not the intention. to fix it we could do this:  String[] words = s.split("[ \t]+"); |
| **Exercise 6.37**  See the modified start() method of SupportSystem in: Code 6.7 and the modified generateResponse() in Responder in: Code 6.8 |
| **Exercise 6.40**  The modifications needed to the Responder class in the project tech-support-complete:  Add a new field:  private HashMap<String, String> synonymMap;  Initialize it in the constructor:  synonymMap = new HashMap<>();  Add this to the generateResponse method right after the if-block:  else {  //check if it is a synonym  String synonym = synonymMap.get(word);  if(synonym != null) {  return responseMap.get(synonym);  }  }  To create an example replace the responseMap.put("crashes","Well....); with: synonymMap.put("crashes", "crash"); |
| **Exercise 6.43**  putIfAbsent **takes two parameters - a key and a value - and inserts the key, value pair into the map only if the given key is not already present in the map.** |
| **Exercise 6.44**  /\*\*  \* Print the words encountered, along with their counts.  \*/  public void printWordCounts()  {  for(String word : counts.keySet()) {  System.out.printf("%s: %d%n", word, counts.get(word));  }  } |
| **Exercise 6.45**  **The getter method in Responder:**  /\*\*  \* Get the set of recognised words.  \* @return the recognised words  \*/  public Set<String> getRecognisedWords()  {  return responseMap.keySet();  }  **The printWordCounts method in WordCounter:**  /\*\*  \* Print the words encountered, along with their counts.  \*/  public void printWordCounts(Set<String> recognisedWords)  {  for(String word : counts.keySet()) {  if(! recognisedWords.contains(word)) {  System.out.printf("%s: %d%n", word,  counts.get(word));  }  }  }  **The call from the SupportSystem class:**  counter.printWordCounts(responder.getRecognisedWords()); |
| **Exercise 6.47**  Keyword examples:  @author @version @param @return  These keywords get special attention so they stand out in the documentation. |
| **Exercise 6.53**   * **Color.BLUE is used to set the pen color in drawSquare()** * **Color.RED is used in drawWheel()** |
| **Exercise 6.54**  **Other colors available in the Color class are BLACK, GREEN, MAGENTA, ORANGE, etc. These are available via both upper-case and lower-case names.** |
| **Exercise 6.56**  **Use the clear() method to clear the whole canvas.** |
| **Exercise 6.57**  /\*\*  \* Draw a triangle on the screen.  \*/  public void drawTriangle()  {  Pen pen = new Pen(320, 260, myCanvas);  pen.setColor(Color.GREEN);  triangle(pen);  }  /\*\*  \* Draw a triangle in the pen's color at the pen's location.  \*/  private void triangle(Pen pen)  {  for(int i = 0; i < 3; i++) {  pen.move(100);  pen.turn(120);  }  } |
| **Exercise 6.58**  /\*\*  \* Draw a pentagon on the screen.  \*/  public void drawPentagon()  {  Pen pen = new Pen(320, 260, myCanvas);  pen.setColor(Color.MAGENTA);  pentagon(pen);  }  /\*\*  \* Draw a pentagon in the pen's color at the pen's location.  \*/  private void pentagon(Pen pen)  {  for(int i = 0; i < 5; i++) {  pen.move(100);  pen.turn(360 / 5);  }  } |
| **Exercise 6.59**  /\*\*  \* Draw a polygon with the given number of sides.  \* @param n The number of sides.  \*/  public void drawPolygon(int n)  {  Pen pen = new Pen(320, 260, myCanvas);  pen.setColor(Color.MAGENTA);  polygon(pen, n);  }  /\*\*  \* Draw a polygon with the given number of side  \* in the pen's color at the pen's location.  \* @param sides The number of sides.  \*/  private void polygon(Pen pen, int sides)  {  for(int i = 0; i < sides; i++) {  pen.move(100);  pen.turn(360 / sides);  }  } |
| **Exercise 6.60**  **See 06-60-spiral:**  /\*\*  \* Draw a spiral in the pen's color at the pen's location.  \*/  private void spiral(Pen pen)  {  // The number of arms.  int arms = 63;  // The current length of the arm being drawn.  int armLength = 3;  // How much longer to make each arm.  int armIncrement = 2;    // Start in the middle.  pen.penUp();  Dimension size = myCanvas.getSize();  pen.moveTo(size.width / 2, size.height / 2);  // Face downwards.  pen.turnTo(90);  pen.penDown();    // Draw arms of increasing length.  for(int arm = 0; arm < arms; arm++) {  pen.move(armLength);  pen.turn(90);  armLength += armIncrement;  }  } |
| **Exercise 6.63**  public void bounce(int numberOfBalls)  {  int ground = 400; // position of the ground line  myCanvas.setVisible(true);  // draw the ground  myCanvas.drawLine(50, ground, 550, ground);  // create and show the balls  HashSet<BouncingBall> balls = new HashSet<BouncingBall>();  for(int i=0; i < numberOfBalls; i++) {  BouncingBall ball = new BouncingBall(50+32\*i, 50, 16, Color.blue, ground, myCanvas);  balls.add(ball);  ball.draw();  }  // make them bounce  boolean finished = false;  while(!finished) {  myCanvas.wait(50); // small delay  for(BouncingBall ball : balls) {  ball.move();  // stop once ball has travelled a certain distance on x axis  if(ball.getXPosition() >= 550 + 32\*numberOfBalls) {  finished = true;  }  }  }  for(BouncingBall ball : balls) {  ball.erase();  }  } |
| **Exercise 6.64**  HashSet is most suitable, because it guarantees that we only have one of each ball in the collection. The HashMap could be used for this as well, but we do not need a map, so it would be a bad choice. |
| **Exercise 6.65**  // create and show the balls  Random random = new Random();  HashSet balls = new HashSet<>();  for(int i=0; i < numberOfBalls; i++) {  Dimension size = myCanvas.getSize();  int x = random.nextInt((int) size.getWidth());   int y = random.nextInt((int) size.getHeight());  BouncingBall ball = new BouncingBall(x, y, 16, Color.blue, ground, myCanvas);  balls.add(ball);  ball.draw();  } |
| **Exercise 6.66+6.67**  See: **06-66-balls-inabox.zip** |
| **Exercise 6.69**  public static final double TOLERANCE = 0.001;  private static final int PASS\_MARK = 40;  public static final char HELP\_COMMAND = 'h'; |
| **Exercise 6.70**  java.lang.Math defines constants for PI and E. |
| **Exercise 6.71**  There are multiple reasons. For instance:   * A name helps a reader of the code to understand the significance or ‘meaning’ of a particular value. * If the value were to need changing, for some reason, the change only has to be made in one place, which guarantees that all uses are updated, without the risk of missing one. * Ensuring that all occurrences of that particular value are typed correctly is difficult, so coding the value in one place solves that problem. * The usages/meanings of two similar values could be quite different. There is a risk that a wrong value might be changed if the values are hard coded rather than names. Giving distinct names makes the distinction clear and reduces this risk. |
| **Exercise 6.72**  The method header is:  static int max(int a, int b); |
| **Exercise 6.73**  The methods in the Math class are static because they implement mathematical *function* operations – their results do not depend on an object's state and they always return the same results given the same arguments. Therefore we do not need an object with state to use them. It is also more convenient that you do not have to create an object before calling the method.  Yes, they could have been instance methods, but that would require that you create an instance of the Math class before you could use the methods. The object would have no useful mutable state, only methods. |
| **Exercise 6.74**  public static long testLoopTime()  {  long startTime = System.nanoTime();  for(int i = 1; i <= 100; i++) {  // Do nothing!  }  long endTime = System.nanoTime();  return endTime - startTime;  } |
| **Exercise 6.75**  a) Yes, you can call a static method from an instance method.  b) No, you cannot call an instance method from a static method (at least not without first creating an object to call it on).  c) Yes, you can call a static method from a static method. |
| **Exercise 6.77**  The main method could look like this:  public static void main(String args[])  {  SupportSystem system = new SupportSystem();   system.start();  } |
| **Exercise 6.79**  Yes, you can use a static field and the constructor(s) of the class to count the number of instantiations. If you have more that one constructor, you would need to increase the count in each of the constructors. This is one way to do it:  public class InstanceCounter {  private static int instanceCount = 0;  public InstanceCounter()  {  instanceCount++;  }  public InstanceCounter(String something)  {  instanceCount++;  }    public static int numberOfInstances()   {  return instanceCount;  } }    It is actually possible to avoid the incrementation in each constructor. You can use an initialiser block which is invoked before the constructor call. This is not a structure that is used very often, and you might be best off without telling your students about it. But if someone should ask you about it, here is how it looks:  public class InstanceCounter {  private static int instanceCount = 0;  {  instanceCount++;   }    public InstanceCounter()  {  }  public InstanceCounter(String something)  {  }  public static int numberOfInstances()   {  return instanceCount;  } } |
| **Exercise 6.81**  **The following solution retains a List<Sighting>. If it is wished to retain the return type of the original version of the method as ArrayList<Sighting> then an ArrayList could be created in the method and initialised with the List returned from the fall to collect.**  /\*\*  \* Return a list of all sightings of the given type of animal  \* in a particular area.  \* @param animal The type of animal.  \* @param area The ID of the area.  \* @return A list of sightings.  \*/  public List<Sighting> getSightingsInArea(String animal, int area)  {  return sightingList.stream()  .filter(aSighting -> animal.equals(aSighting.getAnimal()))  .filter(aSighting -> aSighting.getArea() == area)  .collect(Collectors.toList());  } |
| **Exercise 6.82**  **The modified addSighting method.**  /\*\*  \* Add a single sighting to our records.  \* @param aSighting The sighting object to be added.  \*/  public void addSighting(Sighting aSighting)  {  String animal = aSighting.getAnimal();  ArrayList<Sighting> sightingList =  animalSightings.get(animal);  if(sightingList == null) {  sightingList = new ArrayList<>();  animalSightings.put(animal, sightingList);  }  sightingList.add(aSighting);  // Add details for the spotter.  int spotter = aSighting.getSpotter();  sightingList = spotterData.get(spotter);  if(sightingList == null) {  sightingList = new ArrayList<>();  spotterData.put(spotter, sightingList);  }  sightingList.add(aSighting);  } |
| **Exercise 6.83**  **The modified printSightingsBy method. Note that getOrDefault has been used to ensure that there is always a collection for the for-each loop, even if there are no sightings by the given spotter.**  /\*\*  \* Print all the sightings by the given spotter.  \* @param spotter The ID of the spotter.  \*/  public void printSightingsBy(int spotter)  {  for(Sighting aSighting :  spotterData.getOrDefault(spotter,  new ArrayList<>())) {  System.out.println(aSighting.getDetails());  }  } |
| **Exercise 6.84**  public class NameGenerator {  public String generatorStarWarsName(String firstName,  String lastName,  String careGiversName,  String cityOfBirth)  {  String swFirstName = lastName.substring(0,3) +  firstName.substring(0,2);  String swLastName = careGiversName.substring(0,2) +  cityOfBirth.substring(0,3);  return swFirstName + " " + swLastName;  } } |
| **Exercise 6.85**  Strings are immutable and therefore can not be changed. The method that is called does not change the instance 's' but returns a new object with the string in upper case. The correct way to do it is:  String upperCaseS = s.toUpperCase(); System.out.println(upperCaseS); |
| **Exercise 6.86**  The variable a and b contain values. When these values are passed as arguments to the method, *the values* get copied into the variables i1 and i2. So we now have two copies of the values in a and b. In the method, we then swap the values stored in the variables i1 and i2. This has no effect outside the method as i1 and i2 are local variables in the method. After calling the method the variables a and b will still contain the same values as before the method call. |