|  |
| --- |
| **Exercise 13.1**  The project no longer compiles. The display() method tries to access the private fields of the Post class, which are not accessible from subclasses. This can be corrected by creating accessor methods in Post and calling these from the subclasses.  If we try to compile after these modifications it still does not work. This time it is in the compilation of the NewsFeed class that fails, because it is trying to invoke the display() method on a variable of type Post. But Post no longer has a method called display() and hence it fails. |
| **Exercise 13.2**  The display() method in the Post class is never called. If the object is a MessagePost the display() method in the MessagePost class is called. If the object is a PhotoPost the display() method in the PhotoPost class is called. This is because the dynamic types of the items are used. |
| **Exercise 13.3**  Yes, it behaves as expected by first calling the display() method from the Post class and then calling the display() method in the actual class (dynamic type).  One problem is that you can't enforce the call to the super class' display() method. This means that if you want to create new subclasses you must remember to call super.display().  Another problem is that you can't modify the order of which the different things are printed out. This is discussed further in Exercise 13.7. |
| **Exercise 13.4**  The toString() method can be found on the class Object. It has no parameters and the return type is String. |
| **Exercise 13.6**  The display() method in the MessagePost class:  public void display()  {  System.out.println(message);  super.display();  } |
| **Exercise 13.7**  Give all the fields in the Post class protected accessors.  Then modify the display() method in the MessagePost class to look like this:  public void display()  {  System.out.println(getUserName());  System.out.println(message);  System.out.print(getTimeString());  System.out.println(" – " + getLikesString());  System.out.println(getCommentsString());  }  Account will have to be taken of the exact format of the strings returned by the accessor methods in Post. |
| **Exercise 13.8**  **It adds:** empty**,** peek**,** pop**,** push **and** search**. Note that an** isEmpty **method is already inherited, so** empty **duplicates some inherited functionality.** |
| **Exercise 13.9**  **A Stack should only permit access to the element at the top of the stack. Several of its inherited methods allow access at positions other than the top. For instance,** insertElementAt **and** remove**.** |
| **Exercise 13.11**  See 13-11-zuul-with-transporter |
| **Exercise 13.12**  To implement a Monster and a Player class in the Zuul project it would probably make sense to have a common superclass (Character) that contains the common behavior of the two classes. |
| **Exercise 13.13**  It depends... you could argue for all of the inheritance relations. Which one to choose depends on the current implementation and (if you know) which features you plan to implement in the near future.  If an Item is a superclass of a Character, it would allow you to treat all characters as Items. This means that a Character could pick up another Character which might make sense in some scenarios.  If Character is a superclass of Item, you can treat all items as Characters. If we define Characters as something that can move around, this would allow for items to move around if that is desired.  If the two classes are siblings and have a common superclass (Thing?), you could do a combination of the above solutions. This could allow a character to pick up Things (which means you can pick up both items and characters) and it could also allow for all Things to move around (again, this also applies to Item and Character).  If you don't consider Item and Character to have anything in common you could have no inheritance relations between Item and Character at all. |
| **Exercise 13.14**  Because type-checking is done on the static type of dev (which is Device) the method getName() must be defined in Device. |
| **Exercise 13.15**  To actually execute a method dynamic method lookup is used. This means that it is the method in the Printer class that will be called, because the dynamic type of dev is Printer. |
| **Exercise 13.16**  Yes, it will compile.  The toString() is implemented in the class Object from the Java library. The Object class is always a superclass of all other classes. Hence, when you execute it is the toString() as defined in the class Object that will be called and the return value assigned to the String variable s. |
| **Exercise 13.17**  Yes, this will compile.  The System.out.println() method can take an Object as argument and as argued in 13.13. Student is a subclass of Object. |
| **Exercise 13.18**  Yes, it compiles.  It prints out all the names of the students in the list.  It will run through all the items in the list and call the method System.out.println(st). This method invokes the toString() method of the object - which, because of dynamic method lookup, will call the toString() method in the Student class. |
| **Exercise 13.19**  D must be a subclass of T  T x = new D(); |