Answer 1:

The design patterns that can be used in order to create an RPG game with random monsters and treasures could be any one of the Creational patterns, namely, AbstractFactory, Factory Method, Builder and Prototype.

In case the Abstract Factory pattern is used, we can pass to CreateMaze an object used as a parameter to create the monsters and treasures etc., To change the classes of Monsters or Treasures we can pass in a different parameter and create different mazes with random monsters and treasures.

Factory Methods encapsulate object creation by letting the subclasses decide what objects to create. We can make use of this property and define the necessary functions in the sub classes to create the random monsters and treasure.

Builder Pattern could also be used to create a new maze in its entirety by passing an object to CreateMaze. The operations for the creation of Monsters and Treasures will be passed in the object passed to CreateMaze. Using the concept of inheritance we can add more monsters and treasure as desired.

Finally using Prototype Patterns, we can parameterize the various prototypical room and the associated monsters and treasures. The pattern takes care of copying these objects and adding it to the maze. By replacing the prototypical objects with different ones we can change the maze's composition to have random monsters and treasures.

The Abstract Factory pattern and the Factory Method are more commonly used than the Builder or the Prototype patterns. However using any of these patters can help us in creating the desired RPG game with random monsters and treasures.

Answer 2:

A singleton class is meant to have only one instance, providing a global point of access to it. When a singleton class is subclassed, a lot of problems are encountered. A singleton class has a private constructor. To subclass it, it's private constructor has to be made public or protected. Making this change doesn't really let the class be a singleton anymore as other classes can instantiate it now. Secondly, the implementation of a Singleton is based on a static variable and subclassing would result in all the derived classes sharing the same instance variable.

Answer 3:

The advantage of using a Factory Method Pattern when there is only one Concrete product is that we can easily decouple the implementation of the product from its use. If any additional products need to be added, or if a product's implementation needs to be changed, it will not affect the creator object. As explained initially, this is because the Creator implementation is not tightly coupled to any Concrete product.

Answer 4:

Factory Method	Abstract Factory Pattern
Factory Method creates objects through Inheritance	Abstract Factory Pattern creates objects using object composition
To create objects using Factory methods, we need to extend a class and override a factory method. So basically the subclass is used to create objects	Abstract factory pattern provides an abstract type for creating a family of products. Subclasses determine how the products are produced. To use the factory we instantiate one object and pass it into some code that is written against the abstract type
Factory method is used to decouple the client code from the concrete classes that need to be instantiated.	Abstract factory patters is used whenever a family of products has to be created and it has to be ensured that the clients create products that belong together. The clients are decouple d from the actual concrete classes they use in this too
Used to create a single product so doesn't really need a big interface.	Interface has to change if new products have to be added, which means even the interface of every subclass has to be changed. Also the interfaces tend to be big
Code is implemented in the abstract creator that makes use of the concrete classes the subclasses create	The concrete factories implement a factory method to create their products.

Answer 5:

The Java I/O package is based on the Decorator pattern. It basically uses decorators to add various functionalities.

To convert the words "I know the Decorator Patterns therefore I RULE!" into a lowercase stream, we would have to decorate the InputStream package with the FilterInputStream package. The FilterInputStream package is the package which we would have to decorate with our custom class to do the lowercase conversion.

We will extend the FilterInputStream class and override the read() method.

Code:

```
//Extend the LowerCaseInputStream from the FilerInputStream and implement the read() method
public class LowerCaseInputStream extends FilterInputStream{

        public LowerCaseInputStream(InputStream in){
                super(in);
```

```
	}

	public int read() throws IOException{
		int c = super.read();
		//Convert the character into lowercase
		return (c == -1 ? c : Character.toLowerCase((char) c));
	}

	//Read each character from a byte array and convert it into LowerCase
	public int read (byte[] b, int offset, int len) throws IOException{
		int result = super.read(b, offset, len);
		for(int i=offset; i < offset + result; i++){
			//Convert each byte into lowercase
			b[i] = (byte) Character.toLowerCase((char) b [i]);
		}
		return result;
	}
}

//Main Method
public class InputTest{

	public static void main(String[] args) throws IOException{

		int c;

		try{

	//test.txt file will contain the words "I know the Decorator Pattern therefore I RULE!"
			InputStream is = new LowerCaseInputStream(new
BufferedInputStream(new FileInputStream("test.txt")));

//Read the content of the test.txt file into the InputStream object
//After that, read each of characters from the InputStream and display
//The data would be in lowercase since the read() method of the LowerCaseInputStream object
would be invoked
			while((c= in.read()) >= 0){
				System.out.println((char)c);
			}
			in.close(); //Close the InputStream object
		}catch(IOException e){
			e.printStackTrace();
		}
	}

}
```

The output of this code would give the result "i know the decorator pattern therefore i rule!" as required