1) Java applications
   a) A java application has a special method, called the "main method," which is where the program will start.
      i) Each class can only have one main method.
      ii) You can "tell" eclipse which main method to run
   b) The main method has the syntax below:

```
public static void main(String[] args){
       //do something fun
}
```

   c) Comments are areas of text ignored by the Java compiler.

```
//this is a single-line comment.
/*this comment is a "multi-line" comment.  It
doesn't end until it is terminated by another
asterisk and backslash, like this*/
```

Create a "hello world" program. Now modify it to take an argument (you can add an argument in eclipse by going to Run→Run configurations→arguments and entering the arguments you want) and say hello to whatever word is given as that argument.

---

2) Java/ OOP (Object Oriented Programming) Concepts

   a) **Class**: defines how to make objects; defined fields (variables part of object) and methods (functions of object)
   b) Variables are used to store information. Each variable has a **type** that specifies the kind of information that can be stored in it.
      i) A declaration tells the compiler to reserve memory space for the value and what to call it.
      ii) An assignment statement assigns a value to a variable.

```
int i; //declaration
i = 10; //assignment
String s; //declaration
s = "Hello"; //assignment
```

      iii) If you want to create a constant value (one that cannot be changed later in the program), you can do that by using the keyword "final"; for example,

```
public final int DAYS_PER_WEEK = 7;
```

         (1) (usually a constant is capitalized to distinguish it from variables whose values can be changed)

What happens if you make an array "final"? Can you change its size? Can you change the values inside? Try it out!

---

   c) **Static (class) members**: fields and methods of a class rather than an object; you don't need an instance of the class to use them; only one copy in entire program and accessed thru class name. Declared as "static"
   d) **Parameters** (arguments or inputs of methods) and local variables disappear when method returns.
      i) Formal parameter: declared in heading.
      ii) Actual parameter: what is given when the method is called.

e) **Constructors**: new instances of a class are created by calling a constructor. (Constructors are attached to classes, so they are not instance methods!) (Constructors cannot have a return value, not even void!)
   i) The default constructor initializes all fields to zero (as soon as another constructor is created, the default constructor becomes inaccessible)

f) **Methods**:
   i) Mutators change the fields inside the class
   ii) Observers (getters) return values of fields in the class.
   iii) Creators create instances of the class.
   iv) **Overloading** methods: parameter types are part of method's signature, so 2 methods can have the same name as long as they have different signatures (return value is NOT signature)

g) Values: Java has both primitive and reference types.
   i) **Primitive types**: are stored directly into the variable; not an "object." Can be autoboxed in 6.0! (Autoboxing is automatically wrapping in a "wrapper object" such as Integer)
      (1) int represents an integer (example: 1)
      (2) double represents a floating point number (example: 1.234232)
      (3) boolean represents a true/false value (example: true)
      (4) char represents a character (example: 'c')
   ii) **Reference types**: objects defined by classes or arrays; before these are initialized, they have a special "null" value.
      (1) You can compare primitive types with "==". But always remember to compare objects (including Strings!) with the .equals() method. (in objects, "==" does alias testing—tests whether the two pointers actually go to the same location in memory. That is not actually what you are probably trying to test. Use ".equals()" instead.
      (2) **Strings** are objects, so variables with type String are reference types.
         (a) Useful String methods: charAt, indexOf, toUpperCase, toLowerCase, toCharArray, equals, substring. Look up or try out these functions.
      (3) Wrappers are also objects; they provide functions associated with primitives such as valueOf, and fields like Integer.MAX_INTEGER.

3) Comparisons:
   a) **Identity versus equality**: for primitives, == and != test equality; for reference types, they are identity (alias) tests.
      i) Equality with objects: you should always override the equals( ) method to provide a reasonable definition of equality for your object (its default behavior is to test that the two variables reference the same memory location)
   b) **Order**: Numeric primitives: <, >, <=, >=. They can also be used for reference types corresponding to primitives (Wrapper classes) due to Autoboxing and Auto-unboxing

---

Strings can exhibit unpredictable behavior if you use ==.
Try out the following. Type in the following code in your main method and put a breakpoint right after "boolean b = x==y;". Then use eclipse debugging to look at the value of b. What is it? How about b2, b3, and b4? Was this what you expected?

```
String x = "hi";
String y = "hi";
boolean b = x==y;
boolean b2 = x.equals(y);
x = String.valueOf(x.toCharArray());
boolean b3 = x==y;
boolean b4 = x.equals(y);
```

The reason for this strange behavior is internal optimizations on the compiler—you don't want to be guessing what the compiler is doing, so don't use ==!

Autoboxing: try a similar test with Integers (the wrapper class for ints). What happens? Can you explain why ? (Hint: autoboxing and auto-unboxing!)

---

4) Object-oriented Programming (OOP):

a) Approach: with a big application, break problem into smaller **subproblems**; assign responsibility to each subproblem; keep each interface small.
   i) Each subproblem has a specification: functionality (services code provides) and interface (input code expects and output conditions guaranteed.) Implementation must meet specification.
   ii) **Specification** should be separated from **implementation** ("**data abstraction**") which makes code more modular, easier to maintain, and allows change of implementation w/o changing interface
   iii) OOP encourages data abstraction and modular code.

b) **static**: a static variable provides data abstraction, but means that there is only ONE variable representing state in the class; they are used for representing states across all objects in the class(for instance, the number of objects of the class that have been created)

c) static methods are not associated with an object instance, so they can only access static variables.

d) **Instance variables**: variables which are associated with a particular instance (object) of a class.

e) **Instance methods**: methods which are shared among all instances of same class, but the method can reference instance variables—and therefore is tied to a particular object instance.

f) OOP provides encapsulation and extensibility
   i) Encapsulation: permits code to be used without knowing implementation details
   ii) Extensibility: permits behavior of classes to be changed/ extended w/o having to rewrite code of class (don't need to involve class implementer). Mechanism in Java: inheritance.

5) **Inheritance**: Classes form a hierarchy (tree) with class Object at root.

a) Each class has at most one superclass; each class has 0 or more subclasses.

b) In a subclass, all public methods and fields of the superclass are available. Even overridden methods are available—you can access them using the keyword "super".

c) **Overriding**: method declaration m in subclass B overrides a method m in superclass A if both methods have same signature (same name, same class/instance type, same number/type of parameters, same return type)

d) **"IS-A" relationship**; form hierarchies. Subclasses can add new fields and methods and can override, but CANNOT remove fields. Must specify own constructors. Direct code reuse; indirect code reuse (polymorphism for methods and type compatibility); note: sibling classes are not type compatible with each other; both are w/ parent.
   i) Constructor: automatic call to **super()** if none is given because that's just the way it works! If no super() on first line, compiler automatically calls it there.
   ii) **Final method**: invariant method; cannot be changed by subclasses; static binding.
   iii) **Final class**: cannot be extended (leaf classes)
   iv) **Static methods**' overriding is resolved at compile-time; others are resolved at run-time.

e) **Overriding**: when derived class provides a new method definition with same signature
   i) **Partial overriding**: derived class method invokes base class method; uses super.methodName()
   ii) At compile time, only visible members of static type of reference can appear at right of dot operator (an Employee presently called a Person can't use an Employee method) No automatic downcasts
   iii) Call if (whatever instanceOf ClassName) to avoid **ClassCastException**
   iv) Arrays: will throw a ClassCastException during runtime if declared as Person[]= new Employee[5] and you try to insert a Student

v) **Covariant return types**: return type of subclass only needs to be type-compatible (subclass of) return type of superclass.
vi) Note: no access to grandparent; only to parent.

f) **Shadowing of Variables**: like overriding, but for fields (variables) and BAD
i) When subclass has a variable of same name (type can be same/ different) as superclass; causes you to lose access of the superclass methods. Also variables use static type (often superclass) not dynamic type and is resolved at compile-time.
ii) Can happen in methods with parameters: can only SEE what is defined in static type!

g) Every class is implicitly a subclass of Object (inheritance in Java is a tree)

h) **Abstract classes**: cannot be instantiated.
i) **Abstract method**: has no definition. Any class with an abstract method must be abstract
ii) Any abstract method must be overridden
iii) Abstract classes can have constructors, but cannot be instantiated, only inherited.

i) **Constructors**: cannot be overridden, but can be invoked by subclass with super() reference; this() or super() must be first call in constructor; if nothing is called there is a default call to super().
i) **Order**: sets all fields to 0 or null; starts w/ constructor of initialized object; chains with "this"; super() is called; calls constructor of superclass; initializes fields in superclass; initializes fields in subclass; continues with constructor.

6) Java Types
a) Types: each variable has two types associated with it: the "real type" and the "associated type". The "real type" is just what it sounds like—the true type of the actual object that the variable references. The "associated type" is the variable's declared type.
i) A variable's "associated type" never changes—as soon as the variable is declared, the "associated type" is set.
(1) The associated type determines what methods the Java compiler will let you use. You can only access methods (and fields) which are defined in the "associated type" of the variable.
ii) A variable's "real type" is dependent on the object to which it points

---

Take a look at the following code segment. Parts of it will not compile—which parts? Why? Try running it and find out.

```
Object obj; //associated type is Object
obj = new String("Hello") //real type is String, associated type still Object.
System.out.println(obj.toString);
obj.substring(1);
obj = new Integer(2); //real type is Integer, associated type still Object.
Integer i = obj;
```

---

b) Casting: in the code above, we ran into 2 problems—since the associated type was Object, we couldn't access String methods like "substring". If we wanted to use these methods, we need to have a variable with real type of String. But in a similar conversion ("Integer i = obj") we also ran into problems. Casting solves these problems. It provides a way of "converting" the associated type of a reference.
i) "Widening" (automatic) casting: done automatically by the compiler; casts an object's type to its supertype. For example, say that we have an Animal class and the Dog class extends it. Then since a Dog is an Animal, it is always "safe" to convert a Dog to an Animal and it will be done automatically when necessary.

ii) "Narrowing" casting has to be done manually. A few examples of this casting are shown below. You need to cast to a type which is consistent with the object's real type. If not, you will get a runtime exception.

```
Object obj; //associated type is Object
Integer i = (Integer)obj;

Animal a = new Dog();
Dog d = (Dog)a;
Cat c = (Cat)a; //will compile fine, but throw an exception!
```

Fix the example above so that it compiles and does not throw any exceptions.

---

c) Java **instanceOf** (ex: "if (p instanceOf IntPuzzle)") is a useful function which can test the real type of an object. It is often used to check if downcast succeeds; tests dynamic type

7) **Abstraction**: hiding of unnecessary detail
   a) **Visibility modifiers**: (method that overrides superclass method cannot have more restricted access; can have less restrictive access). Support ADT's abstraction barrier, along with interfaces
      i) Public: accessible to all
      ii) Protected: visible from class and its subclasses
      iii) Default (package visibility): visible from any class in same package
      iv) Private: not accessible outside of class

Create a class Vehicle with method public int numberOfWheels(). You can return anything you want for Vehicle. Then create subclasses Tricycle, Bike, and Car—which return the appropriate number of wheels. If each has a method "ride()" but Vehicle doesn't, can a Vehicle v= new Car() call the "ride()" method? Create a new subclass of Vehicle, StrangeRide, which has half the number of wheels as you assigned to Vehicle. You will need to call numberOfWheels() of the Vehicle class—which you have overridden. Use the "super" keyword.

---

8) **Recursion**: write a description of algorithm and reason out base cases before writing code; separates logic of program from coding details and stops mental fog of syntactic junk.

Try writing Factorial (!) recursively. What is the base case?Recursive case?

---

   a) Approach: any recursive problem can be expressed as iterative. Decide by clearer vs. efficiency
      i) Recursive case: try to find a parameter such that the solution can be obtained by combining solutions to the same problem using smaller values of n
      ii) Base case: Find small values for n for which you can just write down the solution. Base case is "stopping point" of a recursive problem.
      iii) Verify that for any valid value of n, applying the reduction will lead to base case.

Try writing the Fibonacci function recursively. It is very slow…try speeding it up by storing each result fib(i) in an array, and then accessing the array results.

---

   b) Recursion uses **"divide and conquer":** reduce a big problem to smaller problems, solve, then recombine them.

i) Inefficient: very time a function is called, it creates a stack frame (a little slice of memory to store current variables). As recursion continues, memory builds up. Base case is reached, and computation is done back down the stacks. (Have to go through everything 2x)

9) **Arrays:** a special built-in feature of Java; objects which store other types. For example, int[] is an array that stores integers.
   a) **declaration:** objectOrPrimitive[] arrayName;
   b) **initialization:** arrayName = new objectOrPrimitive[sizeOfArray];
   c) **Pros**: random access, knowledge of size; easy to find objects; uncomplicated, fast sorting
   d) **Cons**: Fixed size that must be known in advance; difficult to add/delete
10) **Exceptions**: thrown to indicate something bad happened
   i) IOException: failure to open/read a file
   ii) ClassCastException: when you attempt to cast an object to a type it can't be cast to
   iii) NullPointerException: if tried to dereference a null
   iv) ArrayIndexOutOfBoundsException
   b) Can be handled with a **try/catch block** (catch clauses are called exception handlers)
   c) Exception is an object: it can be defined and thrown (create your own by defining a class that extends another Exception)
   d) The throws clause: throw must either be declared in method header or caught ("throws" mean can, not does, throw). Subtypes of RuntimeException don't need to be declared.
   e) **Handling:** if no handler is found in method, method terminates abruptly and control is passed back to calling method, who tries to handle exception, etc. If NO calling methods can handle exception, entire program terminates w/ error method.
   f) **Finally clause**: can be used to "clean up" files; is called at the end whether or not the exception is called.
   g) It is usually good form to leave exceptions unhandled until main—at that point, always surround your code with a try/catch block.

Fix the example above so that it compiles and does not throw any exceptions.

---

11) Scanner: scanner is a class that provides a useful way of obtaining input. Look through its API to get an idea of its functionality.

Write a class that parses a String where each line represents a person with a firstname, lastname, age, and a boolean representing whether they are a student. Use Scanner to get each line (the hasNextLine() and nextLine()) functions), then for each line, create a new Scanner which parses the line (use next() (for the firstname), next() (for the lastname), nextInt() (for the age), and nextBoolean() (for the student status).

"John Smith 20 true \n Bobby Roberts 10 true \n Samantha Jones 30 false \n Jim Jones 40 false"

Now modify your code to safely handle malformed input by checking each field first with hasNextWhatever() and throwing an Exception if the field is wrong. You can use the sample string below:
"John Smith 20 true \n Bobby Roberts true 10 \n Samantha Jones 30 false \n Jim Jones 40 false"
