All questions are worth 1 point unless otherwise specified. Answers for lists of possible options are marked with *

1) 3pts. Give 3 ways that dynamic typing is different from static typing (in terms of its use for programming or its implementation). Instructor will grade this manually, so disregard automated grade. (3 points)
   Possible answers:
   No type modifier on variable
   Variable can hold objects/data structures of different types over its lifetime
   Object attributes must be looked up dynamically on every access
   Compiler is unable to check and catch many errors statically
   Type safety can only be ensured by checking each access dynamically (at runtime)

2) 3pts. Which languages are statically typed and use static type inference?
   Java, Lua, *ML, *Scala, node.js, PHP, *Go, bash

3) 4pts. Which languages are dynamically typed?

4) 1pt. The "dot" operator in Java/C#/C++ for a method call is really just "syntactic sugar". Write down what this statement translates to in prototype/source code:
   obj.meth(4);
   Possible Answers:
   meth(*obj, 4)
   meth(obj, 4)

5) 1pt. Given this code:

   public class Parent{
       void foo();
   }

   public class Child extends Parent {
       void foo();
   }
code elsewhere:
Parent p = new Child();
p.foo();
If this language uses static dispatch of instance methods, which implementation (Child's or Parent's) is invoked?
Ans: Parent

6) 1pt. Given this code:
public class Parent{
    void foo();
}
public class Child extends Parent {
    void foo();
}
code elsewhere:
Parent p = new Child();
((Child)p).foo();
If this language uses static dispatch of instance methods, which implementation (Child's or Parent's) is invoked?
Ans: Child's

7) 1pt. Given this code:
public class Parent{
    void foo();
}
public class Child extends Parent {
    void foo();
}
code elsewhere:

Parent p = new Child();

p.foo();

If this language uses dynamic dispatch of instance methods, which implementation (Child's or Parent's) is invoked?
Ans: Child
No matter what the type of the variable is or if there is a cast or not, dynamic dispatch ALWAYS looks up the type of the underlying object at runtime and uses its type to determine which method to invoked (here, p holds a Child object so its Child's method implementation that is invoked).

8) 1pt. Given this code:

```java
public class Parent{
    void foo();
}

public class Child extends Parent {
    void foo();
}

code elsewhere:

Parent p = new Child();

((Parent)p).foo();

If this language uses dynamic dispatch of instance methods, which implementation (Child's or Parent's) is invoked?
Answer: Child
No matter what the type of the variable is or if there is a cast or not, dynamic dispatch ALWAYS looks up the type of the underlying object at runtime and uses its type to determine which method to invoked (here, p holds a Child object so its Child's method implementation that is invoked).

9) 1pt. Which line numbers (separated by commas) require dynamic
checking?
class Parent {
    int field1;
    int field2;
}
class Child extends Parent {
    int field2;
}

1: Child c = new Child();
2: Parent p = new Child();
3: ((Parent)c).field1 = 2;
4: ((Child)c).field1 = 2;
5: ((Parent)p).field1 = 2;
*6: ((Child)p).field2 = 2;

==========================
10) 3pts. Note that multiple answers are possible. Given this code:
class Parent {
    static int f1 = 4;
    static int f2 = 7;
}
class Child extends Parent {
    static int f1 = 2;
}
Which of the following stores 0 in all three fields without error?
Parent.f1 = Parent.f2 = 0;
*Parent.f1 = Parent.f2 = Child.f1 = 0;
Static members should always be accessed via the class name in which they are defined (or inherited by).
Parent p = new Parent(); p.f1 = p.f2 = ((Child)p).f1 = 0;
*Parent p = new Parent(); p.f1 = p.f2 = Child.f1 = 0;
Static fields and methods can be accessed through variables that refer to object instance of the class/type in which they are defined (using the class name as is done for Child.f1 is better though as it makes it clear that the field/method is static). Accessing static members through variable/object instances can confuse a reader into thinking the field/method is non-static.
Child c = new Parent(); Parent.f1 = Parent.f2 = Child.f1 = 0;

*Child c = new Child(); c.f1 = c.f1 = ((Parent)c).f1 = 0;
Accessing static members through a variable is allowed (but not preferred). When doing so, the compiler uses the variable type or a cast if any, to determine which field to access.

11) 1pt. Given this code:

class Parent {
    int field1 = 10;
    int field2 = 20;
}

class Child extends Parent {
    int field1 = 5;
    int field3 = 4;
}

Later code:
int val, val2, val3, val4;
val = val2 = val3 = val4 = 0;
Parent p = new Child();
Child c = new Child();
val = ((Parent)c).field1;
val2 = ((Child)p).field1;
val3 = c.field2;
val4 = p.field3;

After these statements, what values do val, val2, val3, and val4 end up with?

Answer:
10, 5, 20, 0//error
Fields (instance or static) are hidden and not overridden -- so all fields inherited or defined are contained in each subtype instance. So inside a Child object there is a field1 inherited from Parent, another field1 defined by Child, a field2 inherited from Parent, and a field3 defined by Child. To get to the ones inherited and hidden by a field defined with the same name (field1) you need to use a cast. So val holds 10 (Parent's b/c of the cast), val2 is assigned only after a dynamic type check that p actually does hold an object of type Child (its a downcast) and because it does, val2 holds 5, val3 holds 20 because Child inherits field2 from Parent (thus gets its own copy), and val4 never gets reassigned because p.field3 is in error (Parent which is the type of P does not have a field3) -- so val4 holds its initialization value which is 0 in this code and the code ends with an error.