List of things not to forget (in no particular order) about Ecstasy:

1. Everything is an object. No “primitive” type system, for example. “Unified” type system.
2. Every object is accessed and manipulated via a reference.
3. A reference has two parts: A type and an identity.
4. It is common to have multiple different references with the same identity, but with different types.
5. All objects have at least three views of themselves, a “private:this”, a “protected:this”, and a “public:this”. There is also a “target:this” that represents the reference that was used to invoke the object.
6. Immutability. It exists. Any object can make itself immutable.
7. Singletons. It exists. A class can be declared as a singleton.
8. There is a difference between a type and a class. A type is just a set of properties and methods. A class is a type *definition*, and can include implementation, and can nest additional classes.
9. Modules. They exist. Built in. A module is the unit of deployment, of versioning, of loading/linking, etc. (Gene: “Including JARJARing.”)
10. There are functions. A function does not require invocation against an object (i.e. a target).
11. A method can be partially (or even completely) bound, by providing arguments for some (or all) of its parameters. When a method is bound by providing arguments for specific parameters, the result is a method that does not contain the parameters that were bound. When a method is bound to a target, the result is a function.
12. A function can be partially (or completely) bound, by providing arguments for some (or all) of its parameters. When a function is bound by providing arguments for specific parameters, the result is a function that does not contain the parameters that were bound.
13. Tuple is built in.
14. Method arguments are a tuple.
15. All methods and functions can return any number of return values. The return values can be obtained as a tuple. A method with 0 return values can be though of a returning a tuple of size 0, which is also known as void.
16. A conditional method is one that returns a tuple whose first value is a boolean, and which the rest of the values in the tuple are only available if the boolean (the first value) is true.
17. “if” and “while” both have support for conditional methods, as in “while (Object o : iter.next)”
18. The base “Object” type has only a few properties and methods. (i) A “type” property of type Type (Q: why isn’t this on Ref?) (ii) a “T as<T>()” method (Q: why isnt’ this on Ref?), and (iii) a “T to<T>()” method. Maybe a protected “meta” property, with things like control over immutability, access to the class info, etc.
19. Automatic compile-time type narrowing by “instanceof”, e.g. “if (s instanceof String) {s.log();}”
20. Every object (reference?) will have a “free” “to<Object[]>()” (or if of class T, any “to<T[]>()”), and a “free” “to<Tuple>()”, because it is possible for the runtime to provide a (read-only) implementation of Object[] and of Tuple (Set Collection Bag etc.) that is “inlined” into each class (or the root class, or whatever), such that the object can pretend that it is an array that contains only itself, or a tuple that contains only itself. (See also Java’s Collections#SingletonList(Object) #SingletonSet(Object) etc.)
21. There is no “this” until after construction. To make this possible, the equiv of “p = new Point(x, y)” is split into several different parts: Validation, Allocation, Initialization, and Post-Construction. Specifically, a function is used to validate the arguments and to specify the necessary initialization, and then the runtime allocates (for some meaning of that term) the object, the runtime initializes the object as specified by the validator, and then the runtime invokes an (optional) event on the object itself, by which point the object is already constructed, and at which point the “this” becomes visible.
22. Language operators map to interfaces. While the term “operator overloading” has been used by other languages, it is more correct to simply understand that all operators are always implemented via interface in Ecstasy, so it is about implementation and not about overloading.
23. An “array” is just another generic collection type. An array is an array of elements, i.e. “interface Array<Element>”. For each location in the array, it is possible to obtain a reference that represents the L-value for that element; this is itself a Ref<Element>.
24. There are a number of ways to “get an array of something”. For example, it is possible to define a constant array in code, such that it is compiled into the XVM file structure as a constant, and that can be obtained as an immutable object at runtime.
25. One way to build a new array is to specify a type, a size, and an initializer function; a data structure of the specified size is constructed and each element is initialized with a return value from the specified function. (Note that any object has an automatic conversion to a function that supplies itself, so an array of ints could be initialized with the function “0”.)
26. Another way to build a new array is to specify a type and an optional desired capacity. The array is initialized as size zero, and elements can be added to the array.
27. One reason why traditional allocation of arrays is not used is that there is no concept of a “default value” for types, like 0 for int and null for all reference types. The main reason is that null is not a sub-type of all reference types, as it is in Java, nor is it a legitimate value for all references (pointers), as it is in C/C++.
28. Null is an enum. Specifically: “enum Nullable{Null}” That means that Null can only be used as a value for something of type Nullable.
29. “String s = Null;” would be illegal. “String | Nullable s = Null;” would be legal, and can be written in the more convenient manner “String? s = Null;”
30. Booleans are also an enum. Specifically: “enum Boolean{False, True}”.
31. There is some implicit set of “imports” that is applied by default (by the compiler) in every source file. This may be used to simplify adoption of the language by importing “x.Nullable.Null” as “null”, “x.Boolean.False” as “false”, and so on.
32. An “enum” is an enumeration of “singleton value” objects. However, it is composed of two separate parts: A super-type that plays the role of the type name, and a set of sub-type values that play the role of the enumerated values. For example, Boolean is the super-type, and False and True are the sub-types. One cannot have an instance of the class Boolean itself, as it is neither False nor True. False is an “instance of” Boolean, as is True. (This is still a “TODO” to figure out the exact mechanics of enums.)
33. All enums implement the Enumeration interface. All values implement the Value interface. All modules implement the Module interface. All packages implement the Package interface. All services implement the Service interface. (etc.)
34. Numeric literals TODO
35. Character and character string literals TODO
36. Byte and binary string literals TODO
37. Type literals: A type can be defined “inline” as a set of methods and properties, either by listing them, or by performing set-based operations on existing types, or any combination thereof. TODO syntax – need to be able to say “classA or classB”; need to be able to say “union/difference of classA and method\_or\_propertyB”;
38. There is a means to specify “this class”, and a means to specify the “enclosing class”.
39. Annotations TODO
40. Security is designed in to the language & runtime. No way to escape the sandbox.
41. Mixins (stateful) and traits (stateless).
42. Services. Concurrency model. “Point of asynchrony.” (There is no “thread”. There is no “synchronized”. There is no “wait”.)
43. Java-like. Should be very easy for Java / C# (and even C/C++) developers to adopt.
44. Conditional compilation & multiple version support built in (and re-inforces modularity and the “correctness verification” of the code.)
45. Cross platform. Option for portable binary with native compilation and execution. Option for native pre-compilation.
46. Decimal type.
47. assert assert:once assert:always assert:test assert:debug