To determine what generics may be more advanced than others, we explored is how generics usage might relate to levels of experience. Under the assumption that higher frequency of usage corresponds to familiarity, we explored frequency of generics usage types to hypothesize the progression developers may take through generics usage types. If usage counts are generally low, we assume generally developers are less familiar with those types of generics. We also assume that the concepts developers are less likely to be familiar with are more advanced concepts.

To create our hierarchy, we first ordered the different generic usage types in order of most to least frequently used for each developer with generics in their repository. We then compared the position of each type of generics usage *across developers* in relation to other types of generics usage. If two generics usage types appeared next to one another in more than half of the repository orderings, we noted a potential ancestor relationship between the two (if A is an ancestor of B, in the context of our work , A and B are learned in succession).

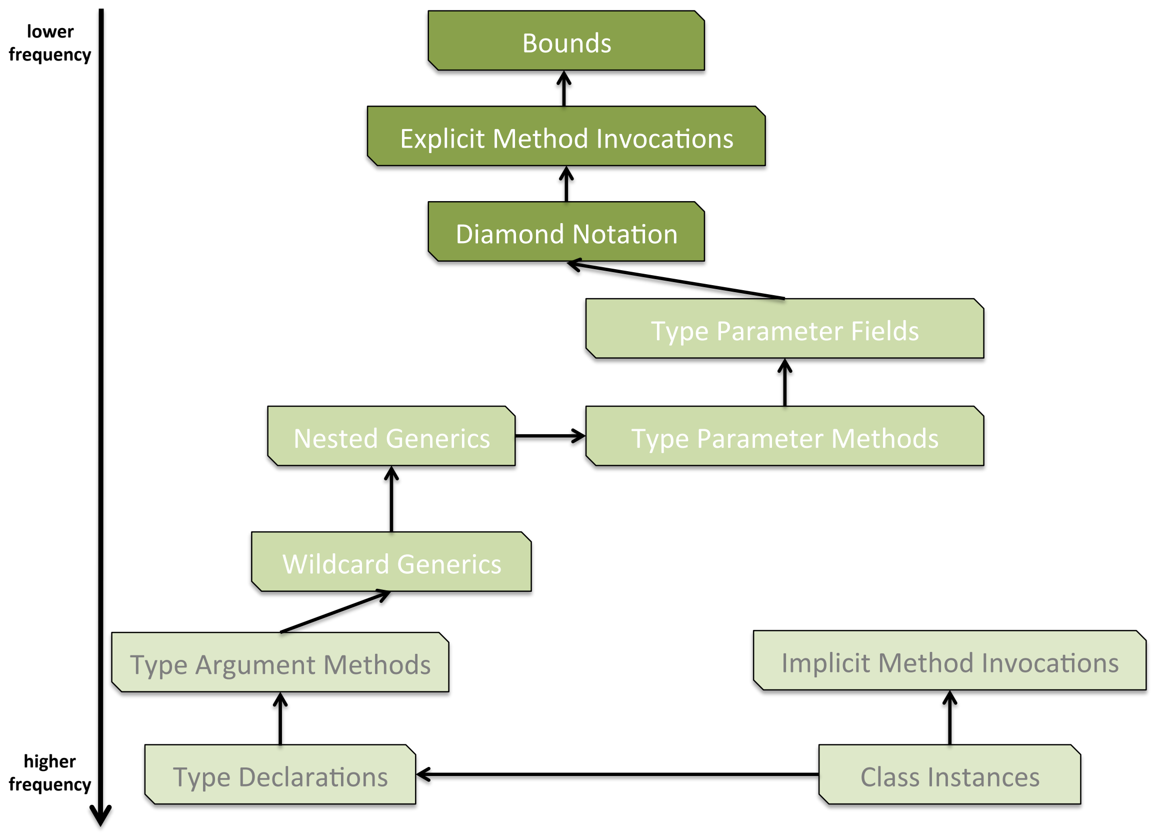

**Figure 1. Proposed generics usage type progression**

Our proposed progression through generics usage types is shown in the Figure 1. The colors indicate the level of generics usage we associate with that type of generics usage (beginner, intermediate, expert). The arrow on the left hand side indicates the direction of code contribution counts. The name for each type of generics usage was determined based on what they are called in ASTParser, the library we used to analyze code, and map to code as follows:

- **Type Declarations** → `public class <T> Box{}`
- **Class Instances** → `List<String> a = new ArrayList<String>();`
- **Implicit Method Invocations** → `HashMap<String, String> a = b.getMap(); getMap()` returns a HashMap
- **Type Argument Methods** → `public List<String> method()`
- **Wildcard Generics** → Any generics that uses the wildcard (?) generic type
- **Nested Generics** → generics in the form of `Observable<Event<T>>` or `Observable<Event<String>>`
- **Type Parameter Methods** → `public T get();`
- **Type Parameter Fields** → `public T t;`
- **Diamond Notation** → `List<String> a = new ArrayList<>();`
- **Explicit Method Invocations** → `HashMap<String, String> a = b.<HashMap<String, String>>getMap();`
- **Bounds** → `public <T extends Comparable<T>> T max(){}`

To determine the cutoffs for the levels in our hierarchy, we looked at the increase in repositories without that usage type. For example, only one repository contained generic bounds, the usage type at the top of our hierarchy.

To validate our groupings, we compared our hierarchy to the questions missed by developers that scored highly on the concept inventory. The concepts at the top half of our hierarchy map to the questions on the inventory missed by top scorers. For example, all top scorers missed the question about explicit method invocations, one of usage types at the top of our hierarchy.

In future work, we will further explore the validity of our hierarchy by creating another hierarchy with a larger set of developers and comparing the two.