



# Visibility, Type Checking and Generics

---

Corky Cartwright  
Department of Computer Science  
Rice University



# Review: What are Language Levels Hiding?

---

- In principle, nothing ...  
Java could have supported a notion of **immutable** classes with the same semantics as the DrJava Intermediate Level.
- But Java is what it is ...
- Transforming DrJava IL code to full Java code:
  - Explicit constructors (initializers!)
  - Explicit accessors
  - Explicit overriding of equals
  - Explicit overriding of hashCode()
  - Explicit overriding of toString()



# Review: Java Constructors

---

The Java virtual machine supports an operation called **new** that allocates an object of specified type in the heap. What is the heap? An area in address space for dynamically allocated (**newed**) objects. **new** also “zeros” out the contents of the allocated object. For each Java type, a zero representation is the “default” value. For object (reference) types, 0 is the null reference (pointer to nothing). For primitive types (**int**, **double**, **boolean**, **char**, ...), 0 is the zero values for numeric types, 0 is **false** for **boolean**, the null character **char**, ...

The empty constructor does nothing! It takes no arguments leaves the default initialization performed by zero-ing in place.

Full Java generates a default empty constructor for any class with no constructor declared in the source (written) code for the class.

Java supports multiple constructors for a class as long as their argument list types (signatures) are different.

With the exception of the empty constructor used in the singleton pattern, you do not need to write any constructors until we move on to Full Java in our assignments (next week).



# Review: Accessors in Full Java

---

In Full Java, accessors are ordinary methods. You must define them for **private** fields if you want other classes to see the field.

What is **private** anyway?



# Simple Visibility

---

- Four visibility modifiers in Java
  - default (package)
  - **public**
  - **private**
  - **protected**
- Visibility modifiers apply to classes and class members
- In simple student programs, default would suffice except for:
  - Java constraint on interface members: must be **public**; the unlabeled members declared in an interface are automatically public.
  - JUnit insists that test classes and the test methods they contain are **public**.
  - Overriding the member of a class cannot narrow its visibility. Some methods inherited from **Object** like **equals** and **toString** are **public**
- DrJava language levels conversion promotes default visibility for methods to **public** except for instance fields which are made **private**. Elementary level prohibits explicit modifiers.



# Full Java Visibility

---

- Java supports an infinite number of distinct name spaces called packages. Each package has a name consisting of a sequence of conventional Java identifiers (names) separated by periods, e.g., `java.lang`. We have been using (and will continue to use) the default package which has no name.
- Libraries and frameworks (except those developed by Sun as part of the Java core libraries) almost always use package names that begin with the name of the organization that created it, e.g., `edu.rice.cs.drjava`.
- Named packages are useful in building production (industrial strength) software but not in simple pedagogic programs.
- We defer discussing the interaction between packages and visibility until later in the course.



# Static Type Checking

---

- A static type system consists of a collection of local rules specifying constraints on the syntactic form of programs. Excluding generics, Java type rules are straightforward and intuitive:
  - Variables and methods always have their declared types.
  - If the context of an expression requires a given type, an expression of some subtype may be used instead. Examples: passing an **Integer** argument to a method that has a parameter of **Object** type.
  - Cast expressions have the type specified in the cast. Casting to a disjoint type is forbidden.
  - Conditional expressions return the least upper bound of the consequent and alternative types.
  - The type (signature) of an overriding method must exactly match the overridden method *except* that the output type can be narrowed (restricted) in an overriding method. Example: see file **IntList.djl** where the output type of **Object forEmptyIntList(EmptyIntList host)** is narrowed in visitors.



# Generics in a Nutshell

---

- A generic class (interface) is a class parameterized by types **T**, **U**, ... most often a single type **T**, e.g. **List<T>**
- Within a generic class, the type parameters can be used like conventional types (almost).
- Outside a generic class, clients always refer to instantiations of the class, e.g. **List<Integer>**
- Generic clients can use their type parameters in such instantiations, e.g., the code in **List<T>** can refer to **EmptyList<T>**
- Static members of a generic class are *not* in the scope of the classes type parameters.
- Generic subtyping is *non-variant* (*invariant?*): **C<S>** **<:** (is a subtype of) **C<T>** iff **S = T**. But it respects erased (ignoring parameterization) class subtyping: **A<S>** **<:** **B<T>** iff **S = T** and **A <:** **B**
- Generics are not available in DrJava language levels.





## Examples:

---

- See `List<E>`, `ListVisitor<E, R>`, ... *etc.*, in `List.java` in the entry for this lecture on course webpage.



# Generics in a Nutshell, cont.

---

- Every type parameter has a fixed upper bound. The default is **Object** but other bounds are sometimes necessary. Bounds are specified using an **extends** clause after the binding occurrence of the type parameter, e.g., **T extends Number**
  - Bounds can refer to the type parameter being bound. Example: look at **Enum<E>** in **java.lang**
  - **class Enum<E extends Enum<E>>**
  - Read about **Enum** at **java.sun.com**
  - Each value of **Enum** type is a singleton class **E** extending **Enum<E>**



# Generics in a Nutshell, cont.

---

A Java class may contain ***polymorphic (generic) methods*** parameterized by types **T**, **U**, ... (typically only one), e.g.,

```
abstract <R> R accept(ListVisitor<E,R> v) ;
```

The ***scope*** of the type parameter is ***restricted*** to the method definition (return type, parameter list, body).

- The class containing a polymorphic method is not necessarily generic.
- The type parameters for a polymorphic method are separately ***bound at each call site***.
- The bindings of polymorphic method type parameters are typically inferred by the Java compiler.
- Study the **accept** methods in the example file **List.java**



# Generics in a Nutshell, cont.

---

Why generics? Much more precise type-checking. Cleaner code without casts. Well-typed programs should never die.

More Advanced Topics:

- Wildcard types which provide co-variant and contra-variant subtyping.
- Details of type inference; in the presence of wildcard types, it is broken. See the recent paper *Java Type Inference Is Broken: Can We Fix It?*,
- Java generics are not first class. `new T()`, `new T[]`, ... (where `T` is a type parameter) are illegal. Parameterized casts are unimplementable (with a few exceptions) and break type safety.
- Can Java generics become first-class in a future version of Java? Perhaps.



# For Next Class

---

- Homework due on Friday. It consists of doing HW6 in Java given a Scheme solution.
- Full answer involves using the visitor pattern.
- Suggestion: do the problem using the interpreter pattern first to write the equivalent of the Scheme functions in the solution that process boolean formulas (represented as abstract syntax trees).
- Convert these methods to visitor objects once you have the program working. If you can't get visitors to work, a flawless interpreter based solution will get 85% credit for the assignment.