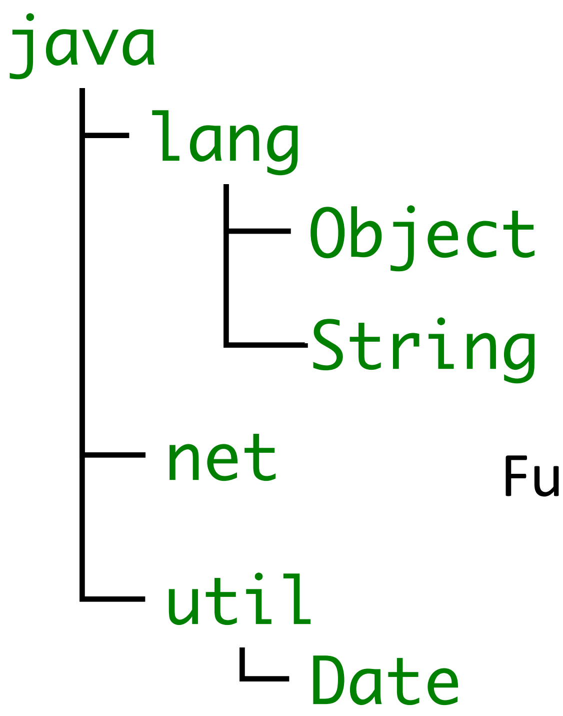
Objectives

- Packages
- Collections
- Generics

PACKAGES

Packages

- Hierarchical structure of Java classes
  - Directories of directories



```
java
├ lang
│   ├ Object
│   └ String
├ net
└ util
    └ Date
```

Fully qualified name: `java.lang.String`

- Use `import` to access packages

Standard Practice

- To reduce chance of a conflict between names of classes, put classes in ***packages***
- Use `package` keyword to say that a class belongs to a package:
  - `package java.util;`
  - *First* line in class file
- Typically, use a unique prefix, similar to domain names
  - `com.ibm`
  - `edu.wlu.cs.logic`

Importing Packages

- Can import one class at a time or all the classes within a package
- Examples:

```
import java.util.Date;
import java.io.*;
```

Import entire package

- * form may increase compile time
  - BUT, no effect on run-time performance

COLLECTIONS

Collections

- Sometimes called *containers*
- Group multiple elements into a single unit
- Store, retrieve, manipulate, and communicate aggregate data
- Represent data items that form a natural group
  - Poker hand (a collection of cards)
  - Mail folder (a collection of messages)
  - Telephone directory (a mapping of names to phone numbers)

Java Collections Framework

- *Unified architecture* for representing and manipulating collections

- More than arrays
  - More flexible, functionality, dynamic sizing

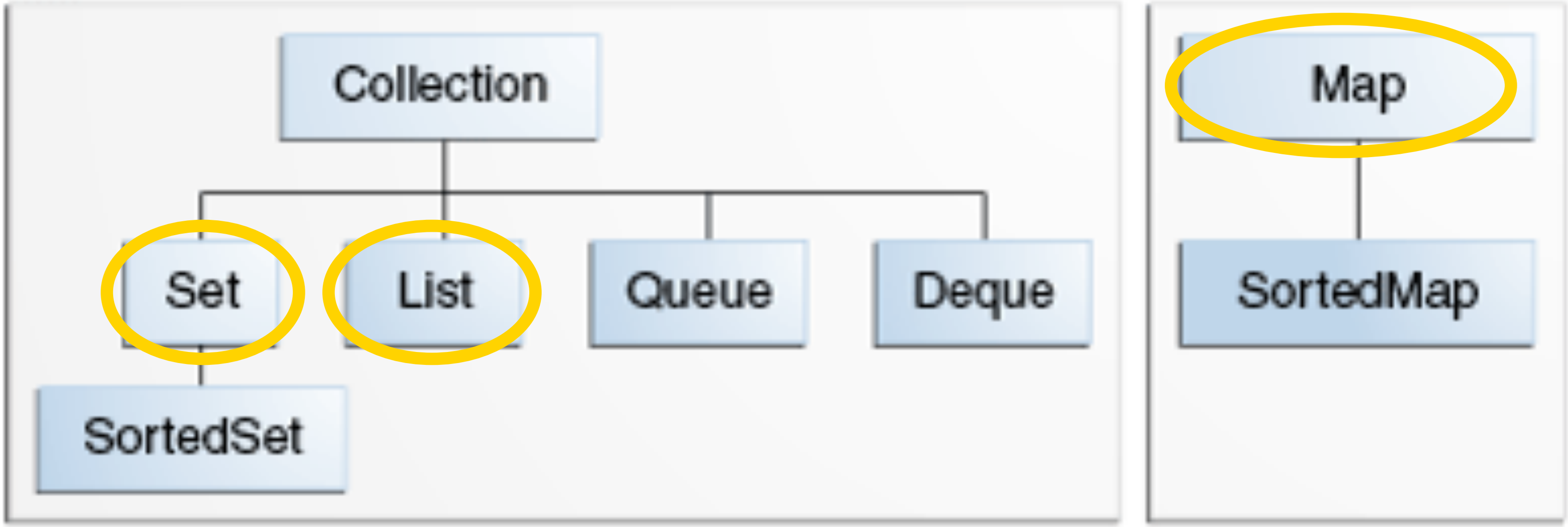
- `java.util`

Collections Framework

- **Interfaces**
  - Abstract data types that represent collections
  - Collections can be manipulated *independently* of implementation
- **Implementations**
  - Concrete implementations of collection interfaces
  - Reusable data structures
- **Algorithms**
  - Methods that perform useful computations on collections, e.g., searching and sorting
  - Reusable functionality
  - ***Polymorphic***: same method can be used on many different implementations of collection interface

Core Collection Interfaces

- Encapsulate different types of collections



GENERICS

Generic Collection Interfaces

- Added to Java in version 1.5
- Declaration of the `Collection` interface:

```
public interface Collection<E> …
```

Type parameter

  - `<E>` means interface is generic for **e**lement class
- When declare a `Collection`, **specify type** of object it contains
  - Make sure put in, get out appropriate type
  - Allows compiler to verify that object's *type* is correct
    - Reduces errors at runtime
- Example, a hand of cards:

Always declare type

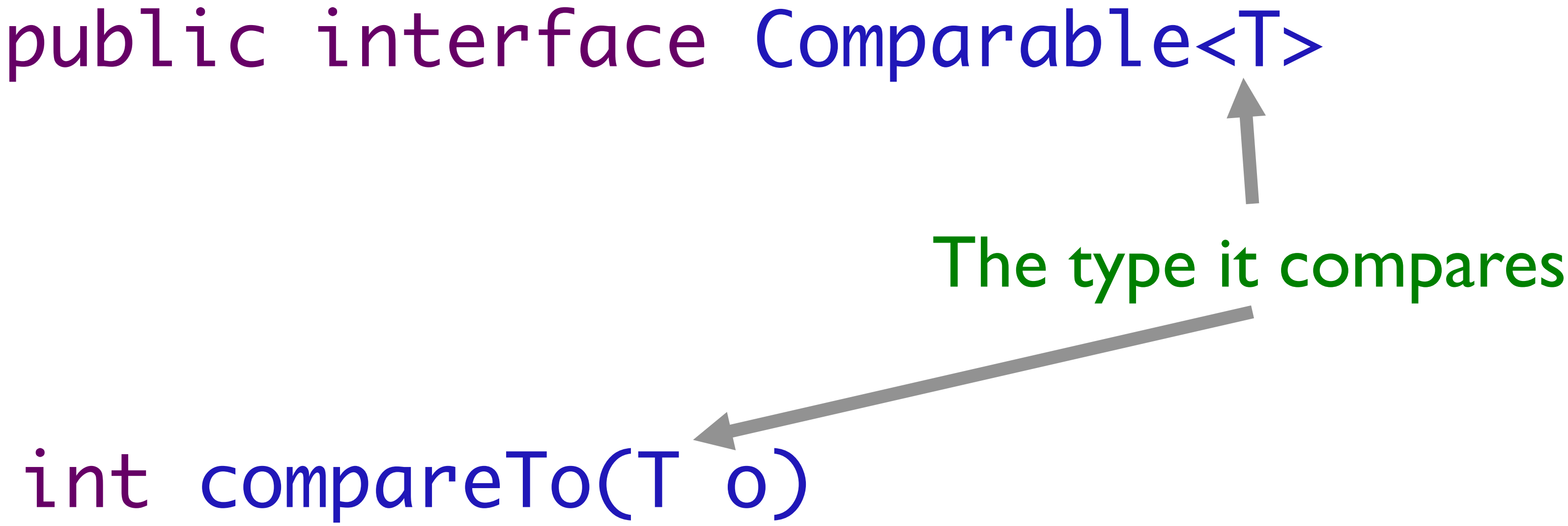
```
List<Card> hand = new ArrayList<Card>();
```

New in Java 7:

```
List<Card> hand = new ArrayList<>();
```

Comparable Interface

- Also uses Generics



```
public interface Comparable<T>
```

The type it compares

```
int compareTo(T o)
```

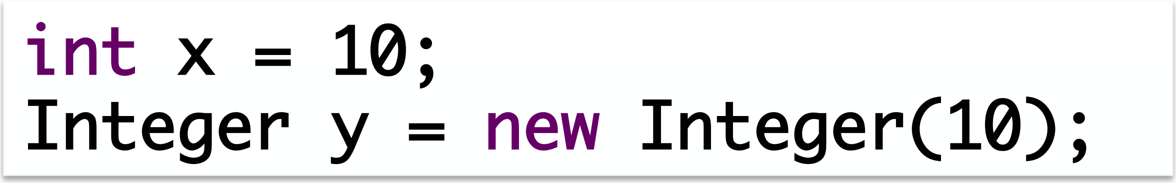
Types Allowed with Generics

- Can only contain `Objects`, not primitive types
  - Autoboxing and Autounboxing to the rescue!
    - Example: If collecting `int`s, use `Integer`

WRAPPER CLASSES

Wrapper Classes

- **Wrapper class** for each primitive type
- Sometimes need an instance of an Object
  - To store in Lists and other Collections
- Include functionality of parsing their respective data types



```
int x = 10;
Integer y = new Integer(10);
```

Wrapper Classes

- **Autoboxing** – automatically create a wrapper object

```
// implicitly 11 converted to
// new Integer(11);
Integer y = 11;
```

- **Autounboxing** – automatically extract a primitive type

```
Integer x = new Integer(11);
int y = x.intValue();
int z = x; // implicitly, x is x.intValue();
```

Convert right side to whatever is needed on the left

LISTS

List

- An *ordered* collection of elements
- Can contain duplicate elements
- Has control over where objects are stored in the list

`List` Interface

- `boolean add(<E> o)`
  - Boolean so that List can refuse some elements
    - e.g., refuse adding `null` elements
- `<E> get(int index)`
  - Returns element at the position index
  - Different from Python: no shorthand
    - Can't write `list`~~`[pos]`~~
- `int size()`
  - Returns the number of elements in the list
- And more!
  - `contains`, `remove`, `toArray`, …

Common `List` Implementations

- `ArrayList`
  - Resizable array
  - Used most frequently
  - Fast

- `LinkedList`
  - Use if adding elements to ends of list
  - Use if often delete from middle of list
  - Implements `Deque` and other methods so that it can be used as a stack or queue

How would you find the other implementations of `List`?

Implementation vs. Interface

Implementation choice only affects performance

- Preferred Style:
  1. Choose an implementation
  2. Assign collection to variable of corresponding **interface** type

```
Interface variable = new Implementation();
```

- Methods should accept interfaces—not implementations

Why is this the preferred style?

SETS

`Set` Interface

- No duplicate elements
  - Needs to determine if two elements are "logically" the same (`equals` method)
- Models mathematical set abstraction

Set Interface

- boolean add(<E> o)
  - Add to set, only if not already present
- int size()
  - Returns the number of elements in the list
- And more! (contains, remove, toArray, …)
  - Note: no get method -- get #3 from the set?

Some `Set` Implementations

- `HashSet`
  - Implements set using *hash table*
    - `add`, `remove`, and `contains` each execute in O(1) time
  - Used more frequently
  - Faster than `TreeSet`
  - No ordering
- `TreeSet`
  - Implements set using a *tree*
    - `add`, `remove`, and `contains` each execute in O(log n) time
  - Sorts