



## Objectives 6

---

## **Collections / Generics**



# Collections

## (Objective 6.1)

---

- Common collection activities include adding objects, removing objects, verifying object inclusion, retrieving objects, and iterating.
- Three meanings for "collection":
  - **collection** Represents the data structure in which objects are stored
  - **Collection** java.util interface from which Set and List extend
  - **Collections** A class that holds static collection utility methods



# Collections

## (Objective 6.1) [contd.]

---

- Four basic flavors of collections include Lists, Sets, Maps, Queues:
  - **Lists of things** Ordered, duplicates allowed, with an index.
  - **Sets of things** May or may not be ordered and/or sorted; duplicates not allowed.
  - **Maps of things with keys** May or may not be ordered and/or sorted; duplicate keys are not allowed.
  - **Queues of things to process** Ordered by FIFO or by priority.
- Four basic sub-flavors of collections Sorted, Unsorted, Ordered, Unordered.
  - **Ordered** Iterating through a collection in a specific, non-random order.
  - **Sorted** Iterating through a collection in a sorted order.
- Sorting can be alphabetic, numeric, or programmer-defined.



## Key Attributes of Common Collection Classes (Objective 6.1)

---

- ArrayList: Fast iteration and fast random access.
- Vector: It's like a slower ArrayList, but it has synchronized methods.
- LinkedList: Good for adding elements to the ends, i.e., stacks and queues.
- HashSet: Fast access, assures no duplicates, provides no ordering.
- LinkedHashSet: No duplicates; iterates by insertion order.
- TreeSet: No duplicates; iterates in sorted order.



# Key Attributes of Common Collection Classes (Objective 6.1) [contd.]

---

- HashMap: Fastest updates (key/value pairs); allows one null key, many null values.
- Hashtable: Like a slower HashMap (as with Vector, due to its synchronized methods). No null values or null keys allowed.
- LinkedHashMap: Faster iterations; iterates by insertion order or last accessed; allows one null key, many null values.
- TreeMap: A sorted map.
- PriorityQueue: A to-do list ordered by the elements' priority.



# Using Collection Classes

---

- Collections hold only Objects, but primitives can be autoboxed.
- Iterate with the enhanced for, or with an Iterator via `hasNext()` & `next()`.
- `hasNext()` determines if more elements exist; the Iterator does NOT move.
- `next()` returns the next element AND moves the Iterator forward.
- To work correctly, a Map's keys must override `equals()` and `hashCode()`.
- Queues use `offer()` to add an element, `poll()` to remove the head of the queue, and `peek()` to look at the head of a queue.



## Sorting and Searching Arrays and Lists

---

- Sorting can be in natural order, or via a Comparable or many Comparators.
- Implement Comparable using compareTo(); provides only one sort order.
- Create many Comparators to sort a class many ways; implement compare().
- To be sorted and searched, a List's elements must be *comparable*.
- To be searched, an array or List must first be sorted.



# Utility Classes: Collections and Arrays

---

- Both of these `java.util` classes provide
  - A `sort()` method. Sort using a `Comparator` or sort using natural order.
  - A `binarySearch()` method. Search a pre-sorted array or `List`.
- `Arrays.asList()` creates a `List` from an array and links them together.
- `Collections.reverse()` reverses the order of elements in a `List`.
- `Collections.reverseOrder()` returns a `Comparator` that sorts in reverse.
- `Lists` and `Sets` have a `toArray()` method to create arrays.



# Generics

---

- Generics let you enforce compile-time type safety on Collections (or other classes and methods declared using generic type parameters).
- An `ArrayList<Animal>` can accept references of type `Dog`, `Cat`, or any other subtype of `Animal` (subclass, or if `Animal` is an interface, implementation).
- When using generic collections, a cast is not needed to get (declared type) elements out of the collection. With non-generic collections, a cast is required:

```
List<String> gList = new ArrayList<String>();  
List list = new ArrayList();  
// more code  
String s = gList.get(0); // no cast needed  
String s = (String)list.get(0); // cast required
```



## Generics [contd.]

---

- You can pass a generic collection into a method that takes a non-generic collection, but the results may be disastrous. The compiler can't stop the method from inserting the wrong type into the previously type safe collection.
- If the compiler can recognize that non-type-safe code is potentially endangering something you originally declared as type-safe, you will get a compiler warning. For instance, if you pass a `List<String>` into a method declared as `void foo(List aList) { aList.add(anInteger); }` the compiler will issue a warning because the `add()` method is potentially an "unsafe operation."
- Remember that "compiles without error" is not the same as "compiles without warnings." On the exam, a compilation *warning* is not considered a compilation *error* or *failure*.



## Generics [contd.]

---

- Generic type information does not exist at runtime—it is for compile-time safety only. Mixing generics with legacy code can create compiled code that may throw an exception at runtime.
- Polymorphic assignments applies only to the base type, not the generic type parameter. You can say

```
List<Animal> aList = new ArrayList<Animal>(); // yes
```

You can't say

```
List<Animal> aList = new ArrayList<Dog>(); // no
```



## Generics [contd.]

---

- The polymorphic assignment rule applies everywhere an assignment can be made. The following are NOT allowed:

```
void foo(List<Animal> aList) { } // cannot take a List<Dog>
```

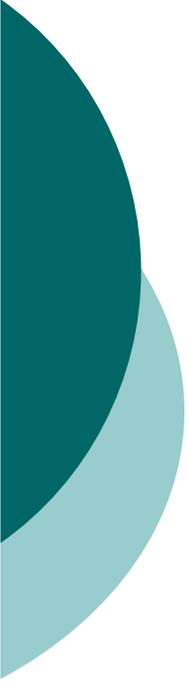
```
List<Animal> bar() { } // cannot return a List<Dog>
```

- Wildcard syntax allows a generic method, accept subtypes (or supertypes) of the declared type of the method argument:

```
void addD(List<Dog> d) { } // can take only <Dog>
```

```
void addD(List<? extends Dog>) { } // take a <Dog> or <Beagle>
```

- The wildcard keyword extends is used to mean either "extends" or "implements." So in <? extends Dog>, Dog can be a class or an interface.

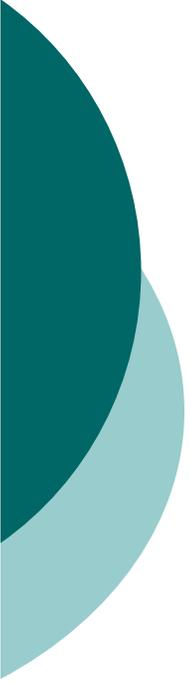


## Generics [contd.]

---

- When using a wildcard, `List<? extends Dog>`, the collection can be accessed but not modified.
- When using a wildcard, `List<?>`, any generic type can be assigned to the reference, but for access only, no modifications.
- `List<Object>` refers only to a `List<Object>`, while `List<?>` or `List<? extends Object>` can hold any type of object, but for access only.
- Declaration conventions for generics use T for type and E for element:

```
public interface List<E>           // API declaration for List
boolean add(E o)                   // List.add() declaration
```



## Generics [contd.]

---

- The generics type identifier can be used in class, method, and variable declarations:

```
class Foo<t> { } // a class
T anInstance; // an instance variable
Foo(T aRef) { } // a constructor argument
void bar(T aRef) { } // a method argument
T baz() { } // a return type
```

The compiler will substitute the actual type.

- You can use more than one parameterized type in a declaration:  
`public class UseTwo<T, X> { }`
- You can declare a generic method using a type not defined in the class:

```
public <T> void makeList(T t) { }
```

is NOT using T as the return type. This method has a void return type, but to use T within the method's argument you must declare the <T>, which happens before the return type.



## Overriding hashCode() and equals() (Objective 6.2)

---

- equals(), hashCode(), and toString() are public.
- Override toString() so that System.out.println() or other methods can see something useful, like your object's state.
- Use == to determine if two reference variables refer to the same object.
- Use equals() to determine if two objects are meaningfully equivalent.
- If you don't override equals(), your objects won't be useful hashing keys.
- If you don't override equals(), different objects can't be considered equal.



# Overriding hashCode() and equals() (Objective 6.2) [contd.]

---

- Strings and wrappers override equals() and make good hashing keys.
- When overriding equals(), use the instanceof operator to be sure you're evaluating an appropriate class.
- When overriding equals(), compare the objects' significant attributes.
- Highlights of the equals() contract:
  - Reflexive: `x.equals(x)` is true.
  - Symmetric: If `x.equals(y)` is true, then `y.equals(x)` must be true.
  - Transitive: If `x.equals(y)` is true, and `y.equals(z)` is true, then `z.equals(x)` is true.
  - Consistent: Multiple calls to `x.equals(y)` will return the same result.
  - Null: If `x` is not null, then `x.equals(null)` is false.



## Overriding hashCode() and equals() (Objective 6.2) [contd.]

---

- If `x.equals(y)` is true, then `x.hashCode() == y.hashCode()` is true.
- If you override `equals()`, override `hashCode()`.
- `HashMap`, `HashSet`, `Hashtable`, `LinkedHashMap`, & `LinkedHashSet` use hashing.
- An appropriate `hashCode()` override sticks to the `hashCode()` contract.
- An efficient `hashCode()` override distributes keys evenly across its buckets.
- An overridden `equals()` must be at least as precise as its `hashCode()` mate.
- To reiterate: if two objects are equal, their hashcodes must be equal.



## Overriding hashCode() and equals() (Objective 6.2) [contd.]

---

- It's legal for a hashCode() method to return the same value for all instances (although in practice it's very inefficient).
- Highlights of the hashCode() contract:
  - Consistent: multiple calls to x.hashCode() return the same integer.
  - If x.equals(y) is true, x.hashCode() == y.hashCode() is true.
  - If x.equals(y) is false, then x.hashCode() == y.hashCode() can be either true or false, but false will tend to create better efficiency.
- transient variables aren't appropriate for equals() and hashCode().