# Java Knowledge

## I. String and Arrays

### String immutability

Once a *string* is created *in memory(heap)*, it cannot be changed. All methods of String do not change the string itself, but rather *return a new String*. If we need a string that can be modified, we should use StringBuffer or StringBuilder (StringBuffer is synchronized, which means it is thread-safe but slower than StringBuilder). Otherwise, there would be a lot of time wasted for Garbage Collection since each time a new String is created. String is supported by a char array. The String class contains 3 fields: char value[], int offset, int count. String is an immutable class in Java. An immutable class is simply a class whose instances cannot be modified. All information in an instance is initialized when the instance is created and the information cannot be modified.

*String pool* is a special storage area in *Method Area*. When a string is created and if the string already exists in the pool, the reference of the existing string will be returned, instead of creating a new object and returning its reference.

In summary, String is designed to be *immutable for the sake of efficiency and security*.

### Create String with double quotes vs. constructor

For example: String a = "abcd"; String b = "abcd"; a == b is true because a and b are referring to the same string literal in the method area. The memory references are the same. When the same string literal is created more than once, only one copy of each distinct string value is stored. This is called "*string interning*".

However, String a = new String("abcd"); String b = new String("abcd"); a == b  is false because c and d refer to two different objects in the heap. Different objects always have different memory references.

### Array length vs. String length()

An *array* is a *container object* that holds a *fixed number* of values of a single type. After an array is created, its length never changes. The array's length is available as a final instance variable length. Therefore, length can be considered as a defining attribute of an array. One array specifies the element type, the number of levels of nested arrays, and the length of the array for at least one of the levels of nesting. An array contains all the members inherited from class Object.

### Check value in an array

There are four different ways to check is an array contains a value.

1. using List: Arrays.asList(arr).contains(value);
2. using Set: set = new HashSet<String>(Arrays.asList(arr)); set.contains(value);
3. using Loop: String s: arr -> s.equals(value); *Most Efficiency*
4. using Arrays.binarySearch()

### Array methods

1. *Print* an array: String arr = Arrays.toString(array); System.out.println(arr)
   If print array directly, it will print the address of the array, c.f. ArrayList, which can be printed directly
2. *Concatenate* arrays: combine = ArrayUtils.addAll(arr1, arr2);

## II. Common Methods

### Equals() vs. hashCode()

1. If two objects are equal, then they must have the same hash code.
2. If two objects have the same hashcode, they may or may not be equal.

The idea behind a Map is to be able to find an object faster than a linear search.

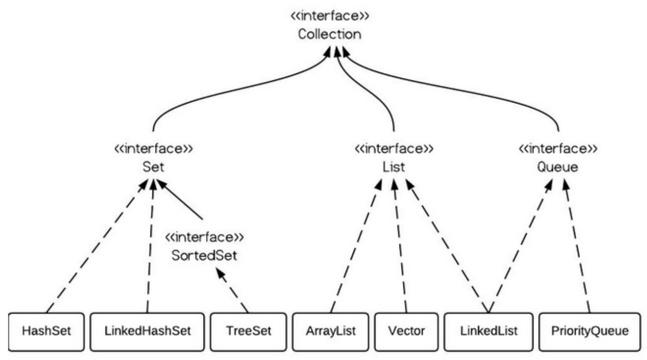
### Recursion vs. Interation

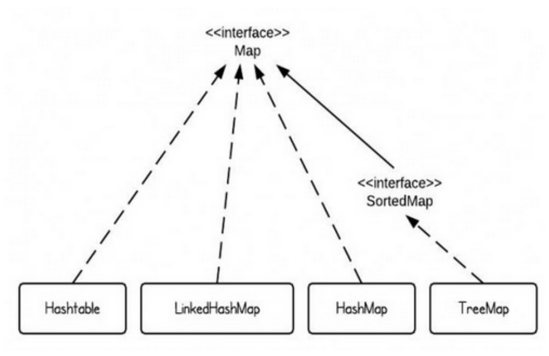
When *recursion*, the computer has to keep *track of the multiplications* to be performed later on. Which characterized by a *chain of operations*. For *iteration*, the computer only need to keep *track of the current values* of the product, whose state can be summarized by a fixed number of variables, a fixed rule that describes how the variables should be updated, and an end test that specifies conditions under which the process should terminate.

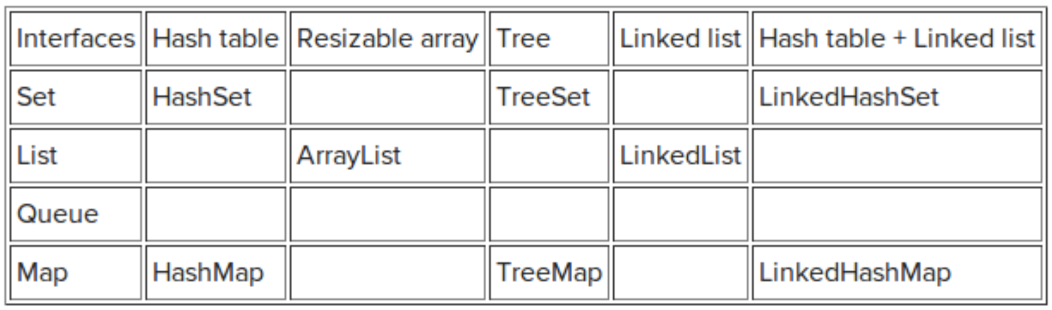
## III. Collections & Generics

### Collection vs. Collections

"Collection" is a root interface in the Collection hierarchy but "Collections" is a class which provide static methods to manipulate on some Collection types.
Class hierarchy of Collection:







| Interfaces | Hash table | Resizable array | Tree | Linked list | Hash table + Linked list |
|---|---|---|---|---|---|
| Set | HashSet | | TreeSet | | LinkedHashSet |
| List | | ArrayList | | LinkedList | |
| Queue | | | | | |
| Map | HashMap | | TreeMap | | LinkedHashMap |

### ArrayList vs. LinkedList vs. Vector

*ArrayList* is implemented as a resizable array. As more elements are added to ArrayList, its size is increased dynamically. It's elements can be accessed directly by using the get and set methods, since ArrayList is essentially an array.
*LinkedList* is implemented as a double linked list. Its performance on add and remove is better than

Arraylist, but worse on get and set methods.
*Vector* is similar with ArrayList, but it is synchronized.
ArrayList is a better choice if your program is thread-safe. Vector and ArrayList require more space as more elements are added. Vector each time doubles its array size, while ArrayList grow 50% of its size each time. LinkedList, however, also implements Queue interface which adds more methods than ArrayList and Vector, such as offer(), peek(), poll(), etc

| | ArrayList | LinkedList |
|---|---|---|
| get() | O(1) | O(n) |
| add() | O(1) | O(1) amortized |
| remove() | O(n) | O(n) |

- ArrayList has O(n) time complexity for arbitrary indices of add/remove, but O(1) for the operation at the end of the list.
- LinkedList has O(n) time complexity for arbitrary indices of add/remove, but O(1) for operations at end/beginning of the List. LinkedList is faster in add and remove, but slower in get

**HashSet vs. TreeSet vs. LinkedHashSet**
*Set* interface extends Collection interface. In a set, *no duplicates* are allowed. Every element in a set must be unique. You can simply add elements to a set, and duplicates will be removed automatically. If you need a fast set, you should use HashSet; if you need a sorted set, then TreeSet should be used; if you need a set that can be store the insertion order, LinkedHashSet should be used.
*HashSet* is Implemented using a hash table. Elements are *not ordered*. The add, remove, and contains methods have constant time complexity *O(1)*.
*TreeSet* is implemented using a tree structure(red-black tree). The elements in a set are *sorted*, but the add, remove, and contains methods has time complexity of *O(log (n))*. It offers several methods to deal with the ordered set like first(), last(), headSet(), tailSet(), etc. TreeSet is much slower because it is sorted.
*LinkedHashSet* is between HashSet and TreeSet. It is implemented as a hash table with a linked list running through it, so it provides the order of insertion. The time complexity of basic methods is *O(1)*.

**HashMap vs. TreeMap vs. LinkedHashMap vs. Hashtable**

*HashMap* is implemented as a hash table, and there is no ordering on keys or values.

*TreeMap* is implemented based on red-black tree structure, and it is ordered by the key.
*LinkedHashMap* preserves the insertion orde

*Hashtable* is synchronized, in contrast to HashMap.

**Set vs. Set<?>**
An unbounded *wildcard type Set<?>* can hold elements of any type, but has two boundaries.
1. Since the question mark ? stands for any type. Set<?> is capable of holding any type of elements.
2. Because we don't know the type of ?, we can't add (method) any element into Set<?>
A *raw type Set* can also hold elements of any type with no restrictions. Thus wildcard type is safe and the raw type is not.

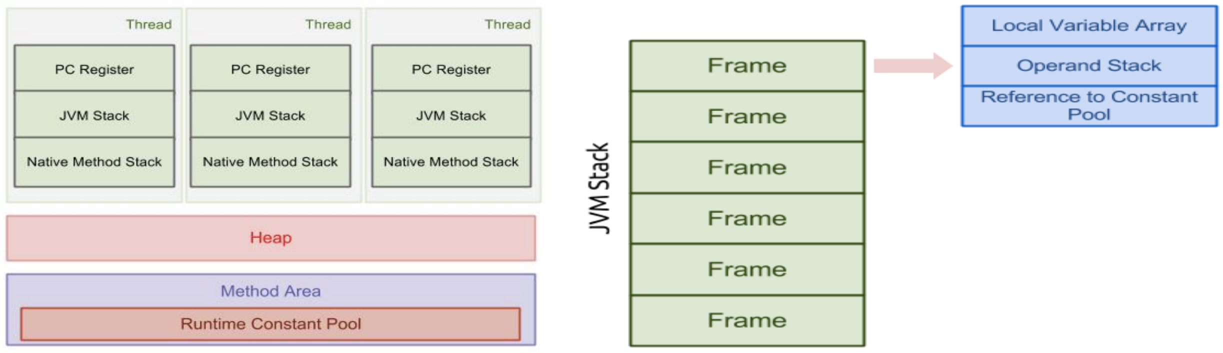
**Comparable vs. Comparator**

*Comparable* is implemented by a class in order to be able to comparing object of itself with some other objects. The class itself must implement the interface in order to be able to compare its instance(s). The method required for implementation is compareTo().

*Comparator* is capable if comparing objects based on different attributes. The method required to implement is compare(). The common use of Comparator is sorting. Both Collections and Arrays classes provide a sort method which use a Comparator. A class that implements Comparator will be a Comparator for some other class. It 1) can be passed to a sort method, such as Collections.sort() or Arrays.sort(), to allow precise control over the sort order and 2) can also be used to control the order of certain data structures, such as sorted sets or sorted maps.

## IV. Compiler and JVM

### JVM run-time data areas



Data Areas for each individual thread include program counter register, JVM Stack, and Native Method Stack. They are all created when a new thread is created.

*Program Counter Register*: it is used to control each execution of each thread.

*JVM Stack*: It contains frames.

*Native Method Stack*: it is used to support native methods, i.e., non-Java language methods.

All threads share Heap and Method Area.

*Heap*: it is the area that we most frequently deal with. It stores arrays and objects, created when JVM starts up. Garbage Collection works in this area.

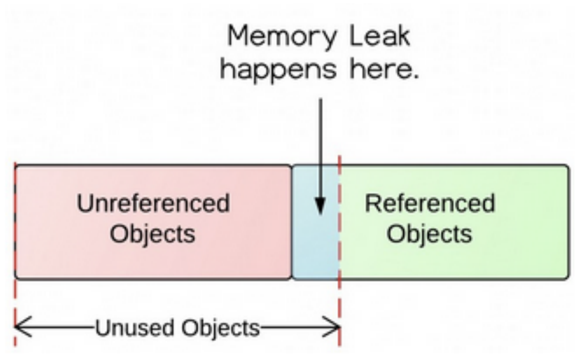
*Method Area*: it stores run-time constant pool, field and method data, and methods and constructors code.

*Runtime Constant Pool*: It is a per-class or per-interface run-time representation of the constant_pool table in a class file. It contains several kinds of constants, ranging from numeric literals known at compile-time to method and field references that must be resolved at run-time.

**Memory Leak**

Objects are no longer being used by the application, but Garbage Collector can not remove them because they are being referenced.

Unreferenced objects will be garbage collected, while referenced objects will not be garbage collected. Unreferenced objects are surely unused, because no other objects refer to it. However, unused objects are not all unreferenced. That's where the memory leaks come from.



1. Pay attention to *Collection classes*, such as HashMap, ArrayList, etc., as they are common places to find memory leaks. When they are declared static, their life time is the same as the life time of the application.
2. Pay attention to *event listeners and callbacks*. A memory leak may occur if a listener is registered but not unregistered when the class is not being used any longer.
3. "If a class manages its own memory, the programer should be alert for memory leaks." Often times member variables of an object that point to other objects need to be null out.

**Java class loaded and initialized**

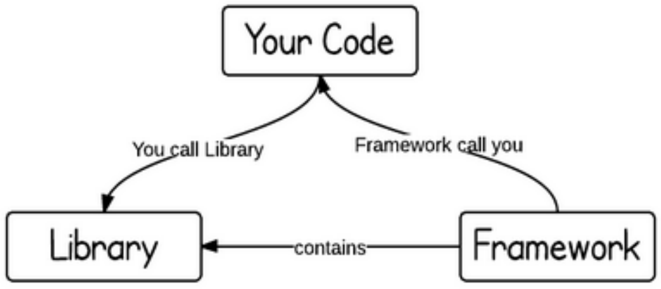
The *linking-like step* (combining source files rom different places and form an executable program) is done when they are loaded into JVM. Basic JVM load classes rule if only loading classes when they are needed. The loading process is recursive. In Java, loading policies is handled by a ClassLoader.
A class is loaded:

- when the new bytecode is executed. For example, SomeClass f = new SomeClass();
- when the bytecodes make a static reference to a class. For example, System.out.

A class is *initialized* when a *symbol* in the class is first used. When a class is loaded it is not initialized. JVM will initialize superclass and fields in textual order, initialize static, final fields first, and give every field a default value before initialization.

**Frames vs. Libraries**

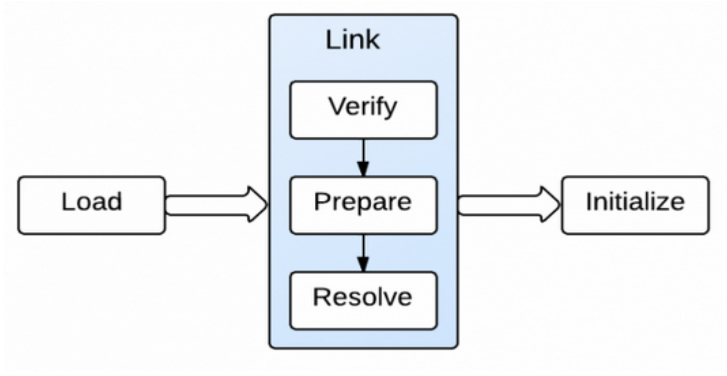
The *key difference* between a library and a framework is "Inversion of Control". When you call a method from a library, you are in control. But with a framework, the framework calls you.



A *library* is just a collection of class definitions. The reason behind is simply code reuse, i.e. get the code that has already been written by other developers. The classes and methods normally define specific operations in a domain specific area.

In *framework*, all the control flow is already there, and there's a bunch of predefined white spots that you should fill out with your code. A framework is normally more complex. It defines a skeleton where the application defines its own features to fill out the skeleton. In this way, your code will be called by the framework when appropriately. The benefit is that developers do not need to worry about if a design is good or not, but just about implementing domain specific functions.
Both of them defined API. To put those together, we can think of a library as a certain function of an application, a framework as the skeleton of the application, and an API is connector to put those together. A typical development process normally *starts with a framework, and fill out functions defined in libraries* through API.

**Classes, main(), bytecode and JVM**
Java programs are built from classes, every method and field has to be in a class. This is due to its *object-oriented feature*: everything is an object which is an instance of a class.
The "*main*" method is the program entrance and it is *static.* "static" means that the method is part of its class, not part of objects. If a method is *not static*, then an object needs to be created first to use the method. Since the method has to be invoked on an object. For the entrance purpose, this is not realistic. Therefore, program entrance method is static. The parameter "*String[] args*" indicates that an array of strings can be sent to the program to help with program initialization.
The *bytecode* is combined with opcode which is not readable, but we can use javap to see the mnemonic form of a .class file.
Before the main method is executed, *JVM* needs to 1) load (Loading brings binary form for a class/interface into JVM), 2) link (Linking incorporates the binary type data into the run-time state of JVM. Linking consists of 3 steps: verification, preparation, and optional resolution. Verification ensures the class/interface is structurally correct; preparation involves allocating memory needed by the class/interface; resolution resolves symbolic references), and 3) initialize the class(assigns the class variables with proper initial values)



This loading job is done by Java Classloaders. When the JVM is started, three class loaders are used

1. Bootstrap class loader: loads the core Java libraries located in the /jre/lib directory.
2. Extensions class loader: loads the code in the extension directories(e.g., /jar/lib/ext).
3. System class loader: loads code found on CLASSPATH.

Finally, the main() frame is pushed into the JVM stack, and program counter(PC) is set accordingly. PC then indicates to push println() frame to the JVM stack. When the main() method completes, it will popped up from the stack and execution is done.

# Head First Java

**Code structure in Java**
One class in a *source file* (with the .java extension) -> methods in a class -> statements in a method. Everything goes in a class. You'll type a source code file, then compile it into a new *class file* (with a .class extension). When running a program, you're really running a class.

**Java loop**
Java has three standard looping constructs: *while, do-while, and for*. The *key* to a loop is the conditional test. A conditional test is an expression that results in a Boolean value (true or false).

**JVM vs. Compiler**
Java virtual machine makes a program run, the compiler just gives a file. Compiler is to stop anything that would never succeed at runtime.

**Objects vs. Class**
*Objects (live on the heap)*: things an object *knows* about itself are called *instance variables*, they represent the *state* of an object (always get a default value); things an object can *do* are called *methods*, they represent the *behavior* of an object.
*Class*: a class describes how to make an object of that class type, it behaves like a template. A class is like a *blueprint*. Everything in Java goes in a class.

**Instance vs. local variables**
*Instance variables* are declared *inside a class* but not within a method, which always get a *default value*. *Local variables* are declared *within a method*, which do NOT get a default value. Method parameters are virtually the same as local variable since they're declared inside the method.

**Objects behave**
A method uses parameters (nothing more than a local variable), a caller passes arguments. If a method has parameters, you must pass arguments of the right type and order. Java is pass by value, which means pass by copy. E.g. int x = 7; void go(int z){z = 0;}; foo.go(x); the argument passed to the z parameter was only a copy of x, the value of x doesn't change.

**Uses of main()**
1. To test your real class
2. To launch/start your Java application

**Variables**
Variables come in two flavors: *primitive and reference*. Variables must have a type and a name. *Primitives* hold fundamental values including boolean, char, byte, short, int, long, float, and double . Each primitive variable has a *fixed number of bits*. *Object reference variable* holds bits that represent a *way to access an object*, it doesn't hold the object itself, but holds something like a pointer, or an address. A reference variable has a value of *null* when it is not referencing any object.

Arrays are objects too. Every element in an array is just a variable (primitive or reference).

**Getter & setter**
A getter's sole purpose in life is to send back as a return value. A setter lives and breathes for the chance to take an argument value and use it to set the value of an instance variable (validate the parameter and decide if it's doable).

**ArrayList vs. Array**
1. Array has to know its size at the time it's created. ArrayList never needs to know how big it should be, it grows and shrinks as objects are added or removed.
2. Put an object in an array need assign it to a specific location. With ArrayList, can specify an index using the add() method.

ArrayList is a class in the core Java Library (the API). A new ArrayList object is created on the heap.

**Java API**
In the Java API, classes are grouped into *packages*. To use a class in the API, you have to know which package the class is in. (full name. e.g. java.util. ArrayList (package name. class name)).
Three reasons why packages are important:
1. Packages help the overall organization of a project or library.
2. Packages give you a name scoping, to help prevent collisions.
3. Packages provide a level of security since you can restrict the code you writhe so that only other classes in the same package can access it.

**Abstract class & method**
*Abstract class*: marking the class as abstract, the class should not be instantiated, the compiler will stop any code from ever creating an instance of that type (stop from saying "new"). you can still use that abstract type as a reference type. An abstract class has virtually no use, no value, no purpose in life, unless it is extended. (A class that's not abstract is called a concrete class.)
*Abstract method*: an abstract method means the method must be overridden (has no body and ends with a semicolon). If you declare an abstract method, you must mark the class abstract as well. You cannot have an abstract method in a non-abstract class (Abstract class can mix both abstract and non-abstract methods). The first concrete class in the inheritance tree must implement all abstract methods.

**Class Object**
Class *Object* is the mother of all classes, it's the *superclass of everything*. Any class that doesn't explicitly extend another class, implicitly extends Object. Object is a *non-abstract class* since it's got method implementation code that all classes can inherit and use out-of-the-box, without having to override the methods. Some of methods in Object can be overridden, some of them are marked final, which cannot be overridden.
The compiler decides whether you can call a method based on the *reference type*, not the actual object type. (e.g. ArrayList<Object> myDog = new ArrayList<Object>(); Dog aDog = new Dog(); myDog.add(aDog); Dog d = myDog.get(0) is false since everything comes out of an ArrayList as a reference of type Object, regardless of what the actual object is).

**Inner object**
An object contains everything it inherits from each of its superclasses, which means every object is also an instance of class Object. There's only ONE object on the heap (the subclass, but it contains both the superclass parts of itself and the parts of itself). Casting an object reference can back to its real type.

**Interface**
A Java interface solves multiple inheritance problems by giving you much of the polymorphic benefits of multiple inheritances without the pain and suffering from the Deadly Diamond of Death. Interface makes all the methods *abstract and public* (like a 100% pure abstract class), and must end in semicolons"()". A class can implement multiple interfaces and extend only one abstract class.
Define an interface: public interface …{}
Implement an interface: public class … extends … implements … {}

**Superclass method invoking**
Call "super. methodName()"

**Stack vs. Heap**
*Objects live on the heap* (all objects live on the garbage-collectible heap), and *method invocations and local variables live on the stack* (two areas of memory). Instance variables live inside the object they belong to (on the heap), local variables also known as stack variables, they're temporary, and live only as long as the method is on the stack. The method lands on the top of a call stack, which is the stack frame, and it holds the state of the method including which line of code is executing, and the values of all local variables. The method at the top of the stack is always the currently-running method for that stack, a method stays on the stack until the method hits its closing curly brace.
If the local variable is a reference to an object, only the variable goes on the stack, the object itself still goes in the heap.
The values of an object's instance variables live inside the object. If the instance variables are all primitives, java makes space for the instance variables based on the primitive type.
An object's life depends entirely on the life of references referring to it. If the reference is considered alive, the object is still alive on the heap.

**Life vs. scope for local variables**
*Life*: a local variable is alive as long as its stack frame is on the stack, until the method completes.
*Scope*: a local variable is in scope only within the method in which the variable was declared. When its own method calls another, the variable is alive, the not in scope until its method resumes. You can use a variable only when it is in scope.

**Constructor**
A constructor looks and feels a lot like a method *without a return type.* It must have the *same name as the class*. It's got the code that runs when you say *"new"* (the code runs when you instantiate an object). Every class you create has a constructor, even if you don't writhe it yourself (the compiler will put in a d*efault constructor* if you don't put a constructor in your class, the default constructor is always a *no-arg constructor*). Always provide a no-arg constructor if you can to make it easy for programmers to make a

working object.
The key feature of a constructor is that it runs *before* the object can be assigned to a reference.
If you have more than one constructor in a class, it means you have *overloaded constructors*.
Constructors can be public, private, or default.
All the constructors in an object's inheritance tree must run when you make a new object. "new" starts the whole constructor chain reaction. The only way to call a *super constructor* is by calling super(), which puts the superclass constructor on the top of the stack. The call to super() must be the *first statement* in each constructor. If you do provide a constructor but you do no t put in the call to super(), the compiler will put a call to super() in each of your overloaded constructors. Each subclass constructor immediately invokes its own superclass constructor, until the Object constructor is on the top of the stack.
Use this() to call a constructor from another overloaded constructor in the same class. this() can be used only in a constructor, and must be the first statement in a construct. Every constructor can have a call to super() or this(), but never both. The keyword "this" is a reference to the current object.

**Object becomes eligible for GC (Garbage Collection)**

1. Last reference dies at end of method
2. Last reference is assigned another object (reprogrammed to a "new" object)
3. Last reference is explicitly set to "null"

**Sort() method**

The ArrayList class does not have a sort() method (array has), you could use Collections.sort() method, which sorts a list of strings alphabetically.

**Generic**

Anytime you see something with *angle brackets* in Java source code or documentation, it means generics. Virtually all of the code you write that deals with generics will be collection-related code. The main point of generics is to let you write type-safe collections.
*Generic Classes*: the class declaration includes a type of parameter.
e.g. public class ArrayList<E> extends AbstractList<E> implements list<E>{
    public Boolean add(E o)}
"E" represents the type used to create an instance of ArrayList.
*Generic Methods:* the method declaration uses a type parameter in its signature.
e.g. public <T extends Animal> void takeThing(ArrayList<T> list)