香港浸會大學
HONG KONG BAPTIST UNIVERSITY

COMP2026

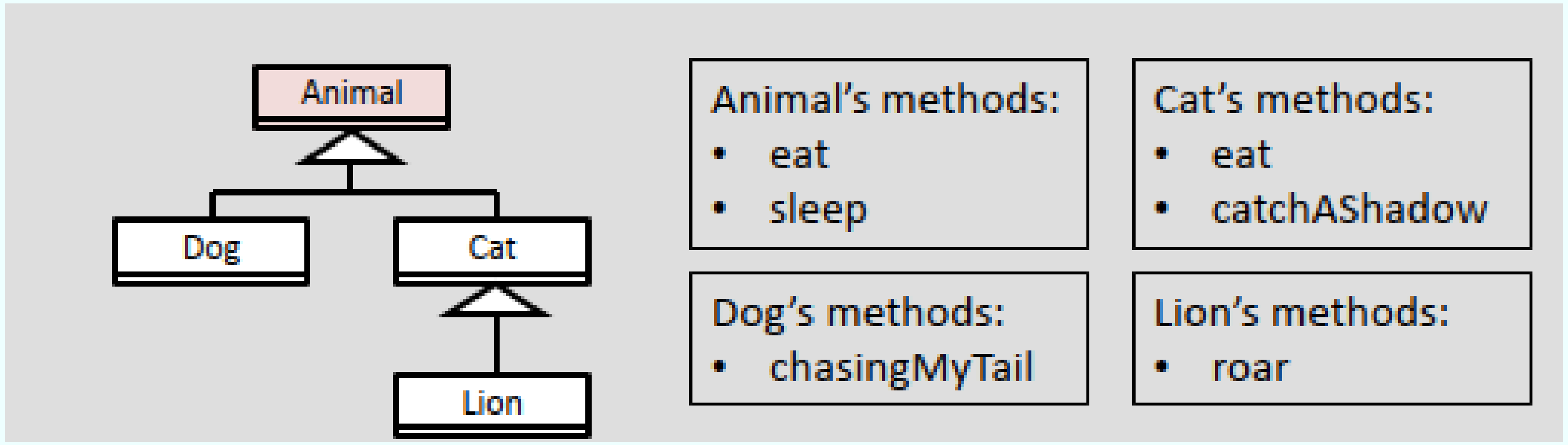
Problem Solving Using Object Oriented Programming

Inheritance - Part 2

Overview

- The `instanceof` Operator
- Access Modifiers
- Visibility of inherited methods (can widen, not narrowed)
- Packages in Java
- Array and ArrayList
- Generic

The instanceof Operator

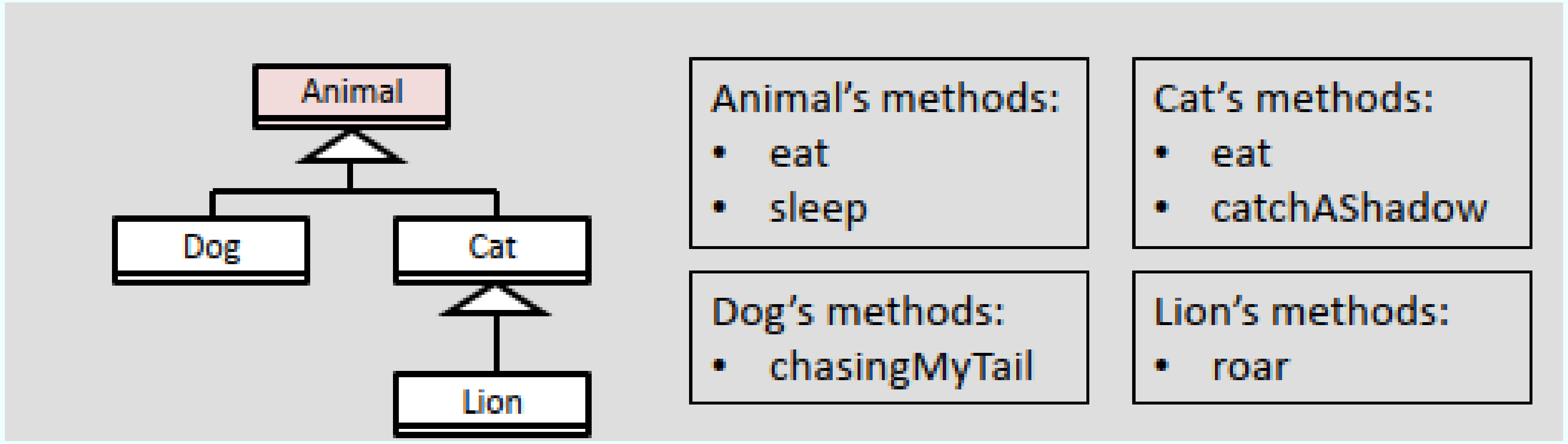


- Java provides the `instanceof` operator, for example:

```
if (aCat instanceof Cat) {
    System.out.println("Yes!  It is a Cat!");
}
```

- All subclass object is an instance of the superclass
- A superclass object may not be an instance of the subclass
- Note: If the object is null, `instanceof` will return false

The instanceof Operator



Animal's methods:
- eat
- sleep

Cat's methods:
- eat
- catchAShadow

Dog's methods:
- chasingMyTail

Lion's methods:
- roar

- `anAnimal instanceof Animal == true`
- `anAnimal instanceof Cat == false`
- `aCat instanceof Animal == true`
- `aCat instanceof Cat == true`
- `aCat instanceof Lion == false`
- `aDog instanceof Cat == false`

Access Modifiers

- Access Modifiers specify what classes “see” or “know about”
  - which methods or instance variables of other classes can you access
  - AKA “visibility modifiers”
- Members of a class can be any one of the four levels of visibility:
  - **private** – never
  - **public** – always
  - **protected** – yes for subclass & same package (not for others)
  - default (package) – yes for same package (not for subclass or others)

private

- private classes: only **inner class** can be declared as private
- private methods:
  - invisible to all other classes
  - never inherited by subclasses
  - never used outside of the class, often called implementation or helper methods because they are written for convenience and code reuse
- private instance variables:
  - same visibility as private methods
  - generally, should make all instance variables private
  - private instance variables are pseudo-inherited – the subclass inherits them but cannot access them directly
    - subclass benefits by using superclass's methods that do have access to all instance variables declared at that level in the hierarchy
    - superclass can provide getters and/or setters to give access to its private instance variables

protected

- protected **classes**: only **inner classes** can be declared protected
- protected **methods**:
    - are strictly visible to:
- all classes (including subclasses) in same package
- protected instance **variables**:
    - same visibility as protected methods
    - protected instance variables are visible to:
        - all classes (including subclasses) in the same package
    - avoid using protected instance variables, except when you want to give subclasses direct access
- protected members are inherited to **subclasses**.
    - Subclass has its own copy of the protected methods/variables.

protected

```
public class Base
{
  private int pri;
  protected int pro;
  public int pub;
  protected void display(){
        System.out.println("in Base");
  }
}
public class Derived extends Base {
  public void showMe(){
    System.out.println(pub); //my pub ok
    System.out.println(pro); //my pro ok
    System.out.println(pri); //Error! does not inherit pri
    display(); //my display, ok
  }
}
```

protected

```
public class Base
{
  private int pri;
  protected int pro;
  public int pub;
  protected void display(){
        System.out.println("in Base");
  }
}
public class Derived extends Base {
  public void showParent(Base b){
    System.out.println(b.pub); //Base's public ok
    System.out.println(b.pro); //We are in the same package, it is ok.
    System.out.println(b.pri); //Error! It is Base's private, not yours!
    b.display(); //my display, ok
  }
}
```

protected but different package

```
package A
public class Base
{
  private int pri;
  protected int pro;
  public int pub;
  protected void display(){
        System.out.println("in Base");
  }
}
```

```
package B
public class Derived extends A.Base {
  public void showMe(){
    System.out.println(pub); //my pub ok
    System.out.println(pro); //my pro ok
    System.out.println(pri); //Error! private is not accessible
    display(); //my display, ok
  }
}
```

protected but different package - Optional

香港浸會大學 HONG KONG BAPTIST UNIVERSITY

```
package A
public class Base
{
  private int pri;
  protected int pro;
  public int pub;
  protected void display(){
        System.out.println("in Base");
  }
}
```

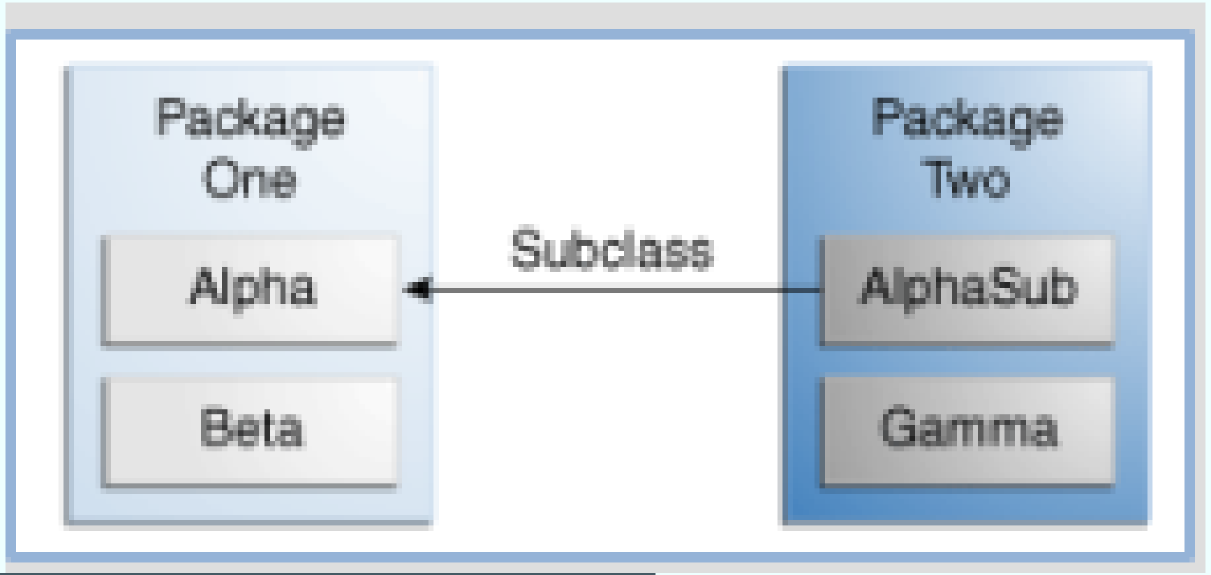
```
package B
public class Derived extends A.Base {
  public void showParent(Base b){
    System.out.println(b.pub); //Base's public ok
    System.out.println(b.pro); //Error! It is Base's protected.
                               //Forbidden if not same package
    System.out.println(b.pri); //Error! It is Base's private, not yours!
    b.display(); //Error! It is Base's protected
  }
```

public

- The **public** modifier means things are visible to all other classes
- **public** classes:
  - visible to everybody
  - good for reusing existing code
  - **public** classes generally go in their own file and file must have the same name as the class
  - generally, make every class **public**
  - exception is implementation or helper classes which you would never want other packages to know about and would never want to reuse – these are internal. We will see some of this later.
- **public** methods:
  - visible to everybody
  - inherited by subclasses
  - exception is implementation or helper methods which you are writing for convenience and code reuse only within the class

Access Controls and Modifiers

- The following table shows where the members (capabilities or properties) of the Alpha class are visible for each of the access modifiers that can be applied to them



Modifier	Alpha	Beta	AlphaSub	Gamma
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	✗
no modifier	✓	✓	✗	✗
`private`	✓	✗	✗	✗

Access Controls and Modifiers

- In general, the following table shows where the members (capabilities or properties) of a class are visible for each of the access modifiers that can be applied to them

Modifier	The Class Itself	Its Package	Its Subclass	Its Superclass
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	✗
no modifier	✓	✓	✗	✗
`private`	✓	✗	✗	✗

Visibility and Inheritance

- Note: Visibility of inherited methods can be widen, but not narrowed

Examples:

1. A `public` method in the superclass, a subclass can only override it as `public`
2. A `protected` method in the superclass, a subclass can only override it as `public` or as `protected`, but cannot be overridden as `private`

Packages in Java

- From the perspective of a class, `ClassA`, it divides classes in the whole execution environment as four different groups:
  - The class itself (that is, members inside `ClassA`)
  - Classes in the same **package**
  - Subclasses of the class
  - All the other classes

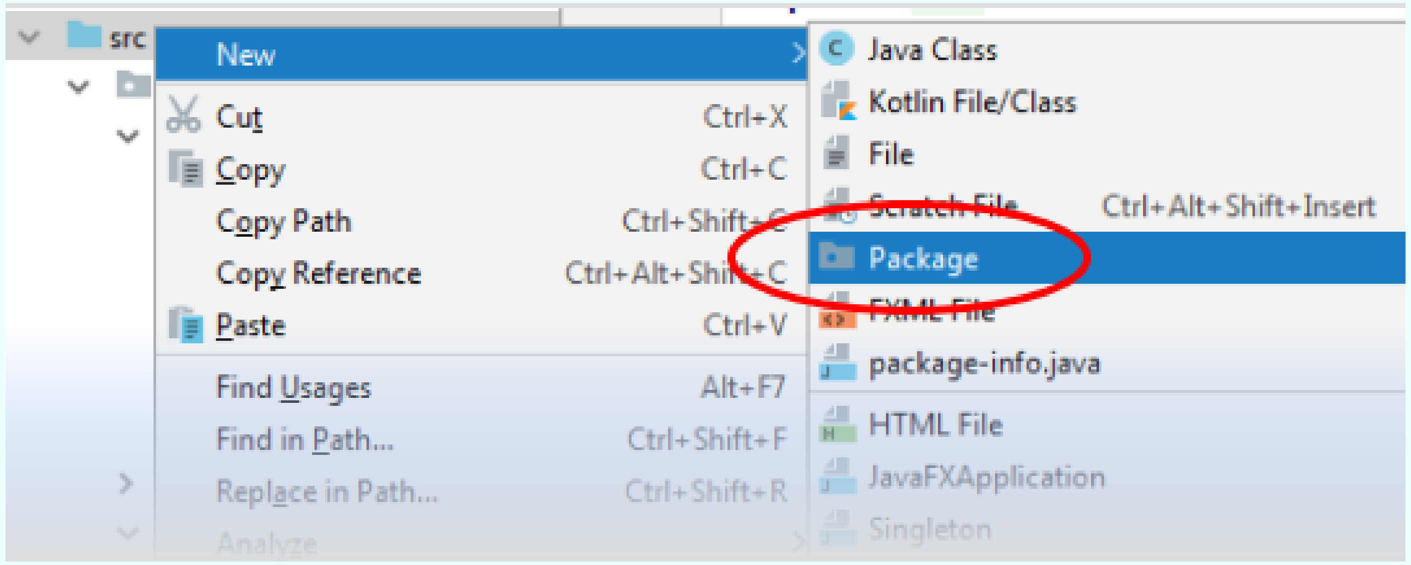
Packages in Java

- A mechanism to group related Java classes
- When a Java project grows bigger, there could be many Java classes
- By dividing classes into different packages, it makes it easier to locate classes that you are looking for
- Classes that work closely with each other are usually grouped together
- With such grouping, related classes can share data/methods easier

Packages in Java

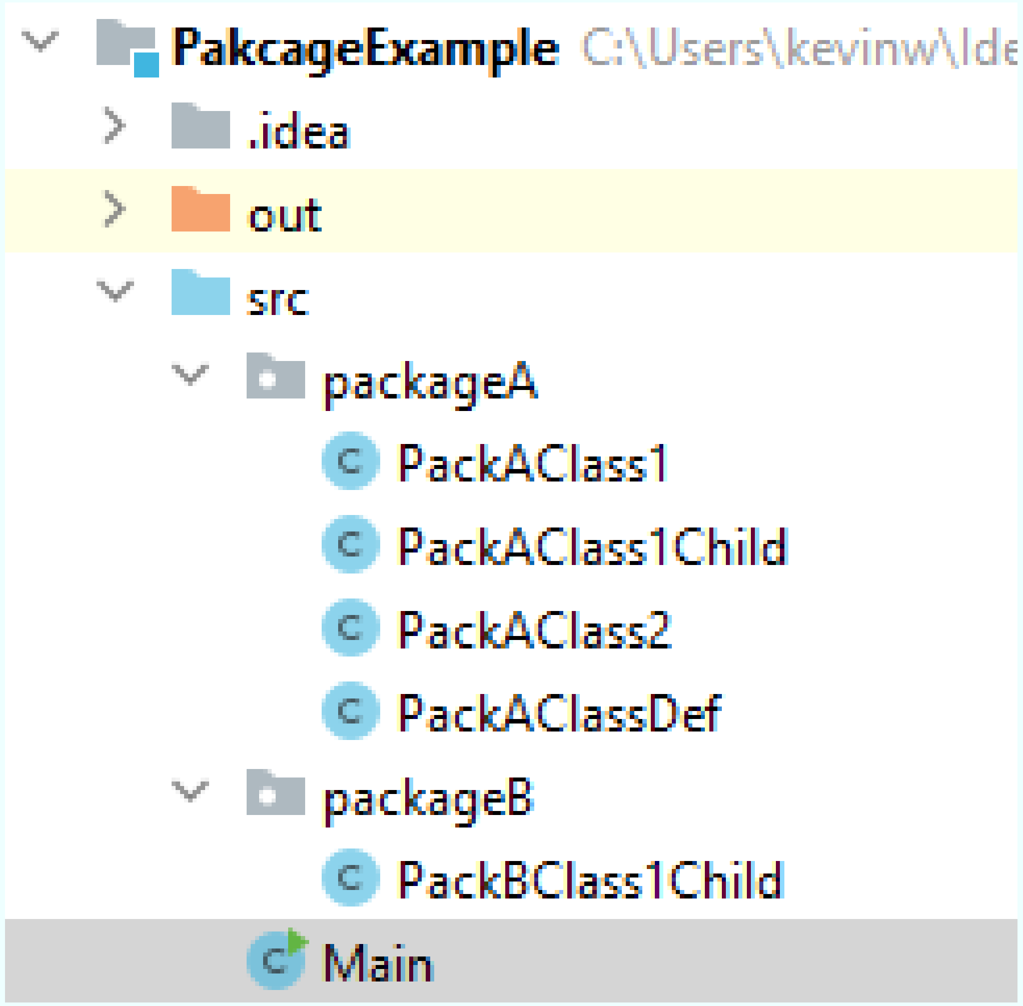
Packages in Java

- A Java package is like a directory in a file system
- In fact, on the disk a package is a directory
- All Java source and class files of classes belonging to the same package are located in the same directory



Packages in Java

- A package folder has to be created to contain all class files of the package
- A package can be in hierarchical, i.e., `packageA` > `subpackageX`, in this case, you need to create a subfolder `subpackageX` under `packageA`.



```
PakcageExample C:\Users\kevinw\Ide
  .idea
  out
  src
    packageA
      PackAClass1
      PackAClass1Child
      PackAClass2
      PackAClassDef
    packageB
      PackBClass1Child
    Main
```

Packages in Java

```java
//file: packageA/PackAClass1.java
package packageA;
import java.util.Scanner;

public class PackAClass1 {
    public int pub = 0;
    int def = 1;
    protected int pro = 2;
    private int pri = 3;
    ...
}
```

- A source file must be started with the keyword `package` which state the name of your package.
- Any import will be place **after** `package`.
- Each source can only in a single package.

Packages in Java

```java
//file: Main.java (not under packageA folder!)
public class Main {
  public static void main(String[] arg) {
    packageA.PackAClass1 myClass = new packageA.PackAClass1();
  }
}
```

- Referencing to a class that is not in your package deserve a full package name reference: `packageName.className`, for each time you refer it.

Import

- Alternatively, you can **import** the class in your source:

```java
//file: Main.java (not under packageA folder!)
import packageA.PackAClass1;
public class Main {
  public static void main(String[] arg) {
    PackAClass1 myClass = new PackAClass1();
  }
}
```

🧑‍🏫 💬 Oh we finally explain what is `import java.util.Scanner;`

Import

- If there are many classes you want to import under the same package, use `*`:

```
import packageA.*;
```

> 🧑‍🏫 💬 Because some classes under different package will share the same name, it is not a good habit to use `*`.

import static (Optional)

- `import static` allows us to import static method from some class, e.g. Math

```
//without static import
double d = Math.sqrt(5) + Math.sin(30);
```

```
import static java.lang.Math.sqrt;
import static java.lang.Math.sin;
double d = sqrt(5) + sin(30);
```

Again you can use `*`. Again it is not a good habit.

```
import static java.lang.Math.*;
```

Shadowing

🧑‍🏫 💬 Warning! Knowing this behavior does not means you should ever use this.

```java
public class A {
  public int var = 5;
}
```

```java
public class B extends A {
  public int var = 4; //same name as parent

}
```

- The variable `var` created in `B` **shadows** its superclass's `var`.

Shadowing

```
public class A {
  public int var = 5;
}
...
public class B extends A {
  public int var = 4; //same name as parent
  void method() {
    System.out.println(var + " " + super.var);
  }
} //print 4 5
```

- `var` by default refers to the field defined in subclass. It is a new variable.
- Use `super.var` to override it.

Shadowing

```
public class A {
  public int var = 5;
  public int getVar() { return var; }
}
...
public class B extends A {
  public int var = 4; //same name as parent
  void method() {
    System.out.println(getVar());
  }
} //print 5
```

- Because `getVar()` is a method of class `A` which knows `A.var` only. Thus it prints 5.

Shadowing

```
public class A {
  public int var = 5;
  }
...
public class B extends A {
  private String var = "abc"; //different visibility and type!
}
```

- Shadowing allows different types of variable because it is a new variable.
- Why shadowing a variable? Can't find a good reason.

Shadowing Rules

Scope	Priority	Explicit Reference
Local variable / Parameter	(1)	-
Field	(2)	`this.x`
Field of Parents	(3)	`super.x`

- If a variable name is defined in local variables or parameter. It shadows Field and superclasses' field with the same name.
- If a variable name is defined in field (without the same local variable/parameter), it shadows superclasses' field.
- To explicitly reference to the objects field: `this.x`
- To explicitly reference to the superclass's field: `super.x`

Superclass's private variable

```
class A {
    private int x;
    protected void getX() { return x;}
}
class B extends A{
    ...
}
```

- Does `B` has `x`?
    - Can we write `this.x` in B?
    - Can we write `super.x` in B?
- Does `B` has `getX()`?
    - Can we write `this.getX()` in B?
    - Can we write `super.getX()` in B?

Superclass's private variable

- The private variable is also inherited from parent **invisibly**.
- The variable x is also copied in the subclass's memory.
- But you cannot access it. Not reading it, not writing it directly.
- `getX()` would still work because your superclass can see `x`. You are using superclass's `getX()` to access `x` indirectly.
- The `x` that the subclass is accessing indirectly, belongs to the subclass.

Superclass's private variable

```java
class Person {
  private final String name;
  Person(String name) {this.name = name;}
  protected String getName() {return name; }
}
class Student extends Person {
  public Student(String name) {
    super(name);   //you can't set name in this class
  }
  public void talk(Student student) {
    System.out.printf("Hi %s, my name is %s. ",
      student.getName(), getName()); //can't access parents name directly
  }
}
```

```java
studentA.talk(studentB);
```

Superclass's private variable

```java
class Person {
  private final String name;
  Person(String name) {this.name = name;}
  protected String getName() {return name; }
}
class Professor extends Person {
  public Professor(String name) {
    super(name);   //you can't set name in this class
  }
  public String getName() {
    return "Dr. " + super.getName(); //without super is a ill-recursion
  }
}
```

- `getName()` shadows superclass `getName()`.
- Use `super.getName()` to access superclass's version! Will talk more details

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

Array and ArrayList

Review Array

```java
public class Container {
  private String[] list = new String[0];
  private void resize(int s) {...}
  public void add(String s) {...}
  public int search(String s) {...}
  public int size() {...}
  public String get(int i) {...}
  public void removeAt(int i) {...}
  public void remove(String) {...}
  public void insertAt(String s, int i) {...}
  public void add(String[] s) {...} //add multiple at the same time
  public void cloneAList(String[] s) {...}
}
```

- Some of the common method you may find with array
- Try to implement them on your own!

Array add

```java
public void add(String s) {
  if (list == null) {
    list = new String[1]; //initialize
    list[0] = s;
    return;
  }
  String[] newList = new String[list.length + 1];
  for (int i = 0; i < list.length; i++)
    newList[i] = list[i];
  newList[list.length] = s;
  list = newList;
}
```

Array search & size & get

```
public int search(String s) {
  for (int i = 0; i < list.length; i++)
    if (list[i].equals(s))
      return i;
  return -1; //return -1 if not found
}

public int size() {
  return list.length;
}

public String get(int i) {
  if (i < 0  || i >= list.length) return null;
  return list[i];
}
```

Array resize

```
private void resize(int s) {
  if (s == list.length) return;
  if (s < 0)
    s = 0;
  String[] newList = new String[s];
  for (int i = 0; i < Math.min(s, list.length); i++) {
    newList[i] = list[i];
  }
  list = newList;
}
```

```
public void removeAt(int index) {
  if (index < 0 || index >= list.length)
    return; //invalid index
  for (int i = index; i < list.length - 1; i++)
    list[i] = list[i + 1];
  resize(list.length - 1);
}
```

Array add - with resize, remove

```
public void add(String s) {
  int size = list.length;
  resize(size + 1);
  list[size] = s;
}
```

```
public void remove(String s) {
  int index = search(s);
  removeAt(index); //do nothing if not found
}
```

Array insertAt and add list

```java
public void insertAt(String s, int pos) {
  if (pos < 0 || pos > list.length) return;
  resize(list.length + 1);
  for (int i = list.length - 1; i > pos; i--)
    list[i] = list[i - 1];
  list[pos] = s;
}
```

```java
public void add(String[] s) {
  for (String i : s)
    add(i);
}
```

Array cloneAList

```
public void cloneAList(String[] a) {
    for (int i = 0; i < list.length; i++)
      a[i] = list[i];
}
```

How can it help us?

- Recall your lab9 Programming exercise...

```java
public class Contact {
  Container list = new Container();
  final String  name;
  public Contact(String name) { this.name = name; }
  public Contact(String name, String phone) {
    this(name);
    addPhoneNo(phone);
  }
  public void addPhoneNo(String phone) { list.add(phone); }
  public String[] getPhoneNos() {
    String[] s = new String[list.size()];
    list.cloneAList(s); return s;
  }
  public void deletePhoneNo(String s) { list.remove(s); }
  public String toString() {
    String output = name + "\n";
    for (int i = 0; i < list.size(); i++)
      output += "[" + i + "] " + list.get(i) + "\n";
    return output;
  }
}
```

Question about this class

Q1. Should I copy it to my assignment/lab/workplace?

> 🧑‍🏫 💬 No need. Java has written similar thing for you, and yet more powerful!

Q2. What if I want to have a list of Contact instead String (like PhoneBook) ? Do I need to rewrite everything?

> 🧑‍🏫 💬 No need. Java has written similar thing for you, and yet more powerful, more **generic**!

ArrayList

- Works very much like array
- A data structure provided by Java
- Manages objects/variables of the **same type**.

Revising ArrayList

```java
public class ArrayListEx1 {
    public static void main(String [] args) {
        List<Person> aList = new ArrayList<>();

        System.out.println("-------------------");
        System.out.println("0: size: " + aList.size());
        aList.add(new Person("Anna"));
        aList.add(new Person("Beatrice"));
        aList.add(new Person("Cathy"));

        System.out.println("-------------------");
        System.out.println("1: size: " + aList.size());
        System.out.println("   alist.get(0): " + aList.get(0));
        System.out.println("   alist.get(1): " + aList.get(1));
        System.out.println("   alist.get(2): " + aList.get(2));
    }
}
```

ArrayList

```
List<Person> aList = new ArrayList<>();
```

- It declares `aList` as a type of `List<Person>`.
- It is fulfilled with the subclass of `List<Person>` - `ArrayList<>`.
- After this line, you should always treat `aList` as a `List<Person>` only, not an `ArrayList<>` because of the inheritance behavior!
- The `<>` and `<person>` will be explained very shortly. Don't worry.

What can an ArrayList do?

Or more relevant, what can a `List<Person>` do? Because `aList` is treated as a `List<Person>`.

Common Accessors (getter)

Return type	Method
`Person`	`get(int index)`
`boolean`	`contains(Person p)`
`int`	`indexOf(Person p)`
`boolean`	`isEmpty()`
`int`	`size()`
`Person[]`	`toArray(Person[] array)`

Common Mutator (setter)

Return type	Method
`boolean`	`add(Person p)`
`void`	`add(int index, Person p)`
`void`	`clear()`
`Person`	`remove(int index)`
`boolean`	`remove(Person p)`
`void`	`sort` *with strange syntax*

With other type..

- What if I want to create a list for String instead of Person?

```
List<String> stringList = new ArrayList<>();
```

- This `<>` symbol specify the type of content to be stored inside the `List`.
- This is about called **generic** (or *template* in other languages)
- Just imagine what you need to change in the class `Container` if you want to make it works for other type, say Person?

Go to the wrong path - changing Container

Container for String

```
public class Container {
  private void resize(int s) {...}
  public void add(String s) {...}
  public int search(String s) {...}
  public int size() {...}
  public String get(int i) {...}
  public void removeAt(int i) {...}
  public void remove(String) {...}
  public void insertAt(String s, int i) {...}
  public void add(String[] s) {...}
  public void cloneAList(String[] s) {...}
}
```

Container for Person

```
public class Container {
  private void resize(int s) {...}
  public void add(Person s) {...}
  public int search(Person s) {...}
  public int size() {...}
  public Person get(int i) {...}
  public void removeAt(int i) {...}
  public void remove(Person) {...}
  public void insertAt(Person s, int i) {...}
  public void add(Person[] s) {...}
  public void cloneAList(Person[] s) {...}
}
```

- Just copy and paste isn't it?
- Why not let the compiler do it for you?

Generic

Generic Version of Container

```java
public class Container<T> {
  private void resize(int s) {...}
  public void add(T s) {...}
  public int search(T s) {...}
  public int size() {...}
  public T get(int i) {...}
  public void removeAt(int i) {...}
  public void remove(T) {...}
  public void insertAt(T s, int i) {...}
  public void add(T[] s) {...}
  public void cloneAList(T[] s) {...}
}
```

Generic

```
public class Container<T> {
    public void add(T[] s) {
      for (T i : s)
        add(i);
   }

   public void add(T s) {
      int size = list.length;
      resize(size + 1);
      list[size] = s;
   }
   ...
```

- `<T>` represent a **generic** type of class that make the program meaningful.
- `for (T i : s)` - imagine you substitute `T` by `String` or `Person`!

Generic - instantiate

- When you instantiate declare your object, you need to specify what `T` is.

```
Container<Person> personContainer = new Container<Person>();
Container<String> stringContainer = new Container<String>();
```

- Java is smart enough to infer the type for you during initialization, i.e.

```
Container<Person> personContainer = new Container<>();
Container<String> stringContainer = new Container<>();
```

❌ Omit T in the type is not allowed, however

```
Container<> personContainer = new Container<Person>(); //error
Container<> stringContainer = new Container<String>(); //error
```

▶

List

- Container example isn't not perfect as we are violate some Java rules in the implementation - creating generic array.
- We have no plan to drill into that!
- But you get the idea
  1. These operations are rather standard, regardless what type it is
  2. A generic structure is needed
  3. List helps a lot!

```java
List<Person> aList = new ArrayList<>();
```

List on primitive type

- List only accept `Class` as a type, not primitive type like `int`, `double`, `float`.
- We use the corresponding Class when primitive data is needed

```
List<int> intList  = new ArrayList<>(); //Error! Not allowed!
List<Integer> intList = new ArrayList<>(); //OK
```

- So then when playing with intList, you are expected to supply `Integer` object and retrieve `Integer` object.

Auto boxing and unboxing

- Auto **Boxing** and **Unboxing** will help you reduce the work

```java
List<Integer> intList = new ArrayList<>();
Integer iObj = new Integer(5); //construct an object
intList.add(iObj);
```

can be rewritten as

```java
List<Integer> intList = new ArrayList<>();
intList.add(5); //auto convert for you
```

```java
Integer iObj = intList.get(0);
System.out.println(iObj.intValue() * 10);
```

rewritten as

```java
System.out.println(list.get(0) * 10);
```

Auto boxing and unboxing

- Autoboxing/unboxing: Automatic conversion between the primitive types and their corresponding object wrapper classes

Primitive Types	Object Wrapper Classes
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

- Note: String is NOT a primitive type!

Errata

- page 11: the `b.display` should be error. Make this page Optional
- page 22: remove `()` from import, remove the package name in the program.