EE3206
Java Programming and Applications

Lecture 3
Inheritance and Polymorphism

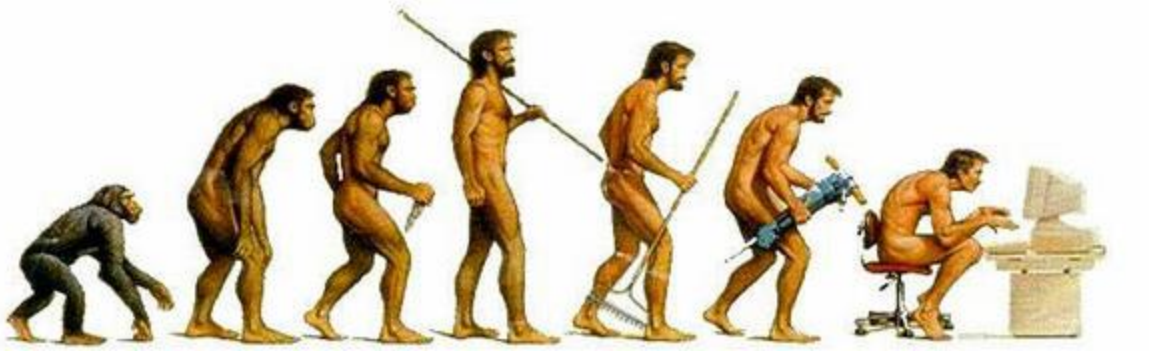
Mr. Van Ting, Dept. of EE, CityU HK

Intended Learning Outcomes

- To derive a subclass from a superclass through inheritance.
- To invoke the superclass's constructors and methods using the super keyword.
- To override methods in the subclass.
- To distinguish differences between overriding and overloading.
- To understand object casting and explain why explicit down-casting is necessary.
- To understand polymorphism and dynamic binding.
- To restrict access to data and methods using the protected visibility modifier.
- To declare constants, unmodifiable methods, and nonextendable classes using the final modifier.

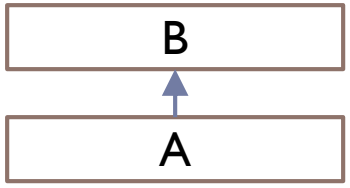
Software Evolution

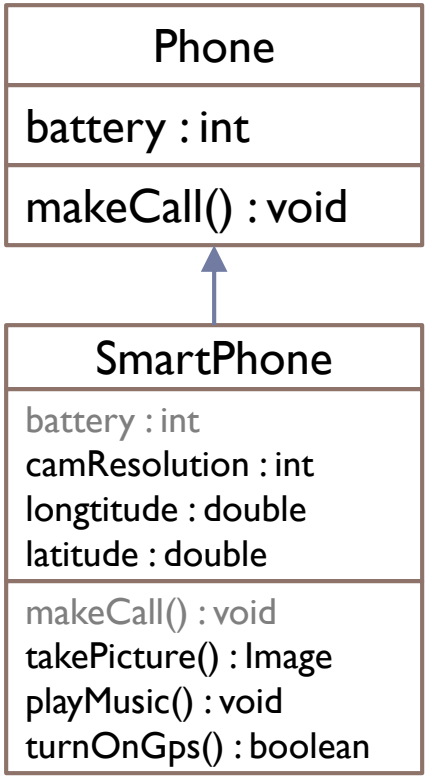
- We use a class to define objects of the same type
- We use objects to represent real world entity
  - identity, state and behavior
- Software requirements change from time to time, so we need to "upgrade" the software's component – class/object.
  - Win98 >> Win2000 >> WinXP >> Vista >> Win7 >> Win8 >> Win 10
  - Instead of start a completely new development from scratch for every next revision, new features can be added gradually to the current version (extending).



Inheritance

- In Java, one class (A) can inherit all the members (i.e. data fields and methods) from another class (B)
- Relationship between A and B is described as Child (A) and Parent (B)
  - A and B are also known as Subclass and Superclass
- Reusability
  - A parent method can be reused (without re-coding) by all subclasses
- Reduce Complexity
  - Subclass only needs to implement the difference between itself and its parent





Extending a Class

- Use the keyword extends
  - class SmartPhone *extends* Phone { … }

- In the previous example, SmartPhone inherits everything (only non-private) from Phone. You may further:
  - Add new fields and methods
    - Some properties the parent do not have
    - Some behaviors the parent do not have
  - Override the methods inherited from the superclass
    - i.e. rewrite the method defined in the parent
    - Usually happen when the inherited method is not appropriate if it applies to the subclass

Method Overriding

```
public class B {
  int b = 10;
  // mean to print all fields
  public void displayAllVars(){
    System.out.println("b=" + b);
  }
}
public class A extends B {
  int a = 20;
  // all fields here include a and b
  public void displayAllVars(){
    System.out.println("b=" + b);
    System.out.println("a=" + a);
  }
}
```

Overriding Vs. Overloading

- Although these two terms look similar, they describe two different Java features:

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
  }
}

class B {
  public void p(int i) {
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

Same signature

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
  }
}

class B {
  public void p(int i) {
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

Different signature

Using the Keyword - *super*

- Similar to the keyword **this**, the keyword **super** can also be used to refer to a member (method or data field) of the superclass:
  - super.methodName();
  - super.name = "Peter";
- You can rewrite the class A in p.6 as below:

```
public class A extends B {

  int a = 20;
  public void displayAllVars(){
    super.displayAllVars(); // calling B's displayAllVars()
    System.out.println("a=" + a);
  }

}
```

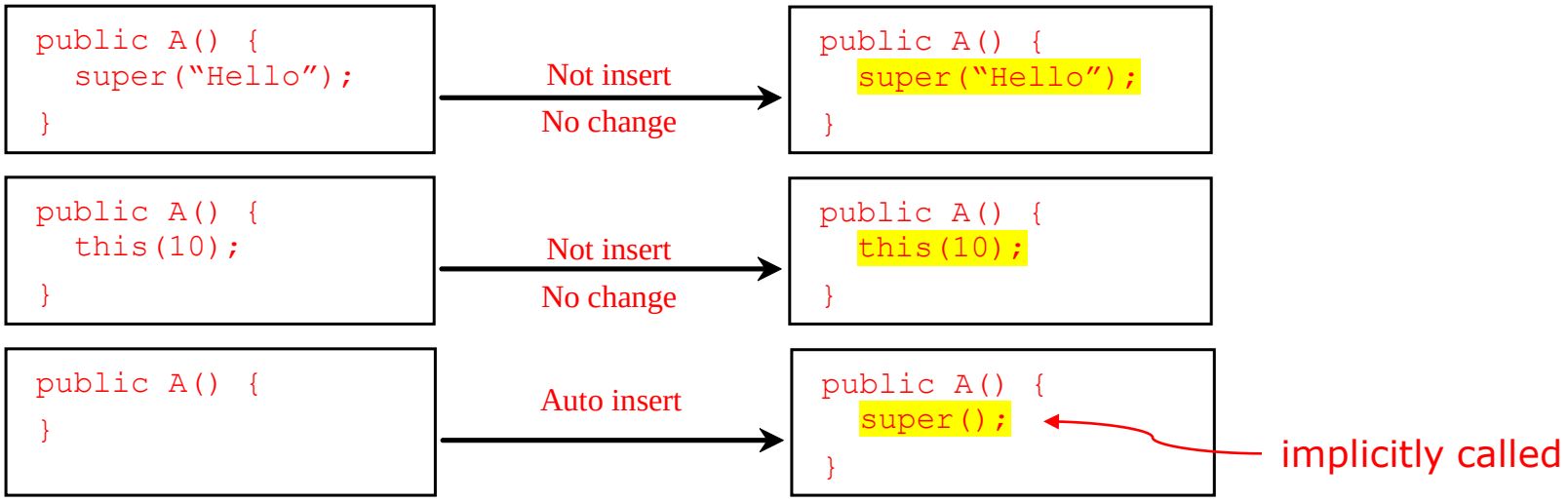
Exercise: Construction Order

```
class B {
    int b = 10;
    public B() {
        b = 20;
    }
}

class A extends B {
    int a;
    public A() {
        a = super.b;          // what is the value of a??
    }

    public static void main(String[] args) {
        A obj = new A();
        System.out.println(obj.a);   // what is the output?
    }
}
```

ConstructorChain

Construction Order

- A constructor may invoke an overloaded constructor (using keyword this) or its super-constructor (using keyword super).
  - Because (constructor) methods in subclass may manipulate the data fields in superclass, Java always runs superclass's constructor before the subclass one.
  - To ensure this happens, the first line of a constructor must either be
    this(...); or *super(...);*
    If none of them is invoked explicitly, the compiler automatically puts *super();* at the first line.



Constructor Chaining

- Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain. This is called constructor chaining.

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

1. Start from the main method

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

2. Invoke Faculty constructor

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

3. Invoke Employee's no-arg constructor

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

4. Invoke Employee(String) constructor

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

5. Invoke Person() constructor

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

6. Execute println

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

7. Execute println

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

8. Execute println

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

9. Execute println

Class Exercise

- Is there any problem with the code?

```
public class Apple extends Fruit {
  public static void main(String[] args) {
    Apple myApply = new Apple();
  }
}

class Fruit {
  public Fruit(String name) {
    System.out.println("Fruit's constructor is invoked");
  }
}
```

The Root Type - *Object* Class

- Every class in Java is descended from the java.lang.Object class. If no inheritance is specified when a class is defined, the superclass of the class is Object.

```
public class Circle {
  ...
}
```

Equivalent

```
public class Circle extends Object {
  ...
}
```

- Every class inherits methods from Object class. There are a few useful methods provided by this class:
  - toString() - return a string representation of the object.
  - hashCode() - return a hash code that uniquely identifies the object
  - equals(Object obj) - indicates whether some other object is "equal to" this one

```
Circle c = new Circle();
System.out.println(c.toString());
```

Polymorphism

- Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
- The only possible way to access an object is through a reference variable.
  - `Circle c = new Circle();  // access Circle's radius via c.radius`
- A reference variable can refer to any object of its declared type or any subtype of its declared type. Let's assume both *Circle and Triangle extend Shape*:
  - `Circle c = new Circle();  // created a Circle`
  - `Shape s = c;  // reference to subtype (up-casting)`
  - `s = new Triangle();  // reseat s reference to a Triangle`
  - `Triangle t = (Triangle) s;  // explicitly casting (down-casting)`
- The type of the reference variable would determine the methods that it can invoke on the object. You can only call methods declared by the reference type no matter what methods are available in the pointed object.

Dynamic Method Binding

- In the previous example, *Shape s* points to different types of objects at different moment. When calling a method from the reference s such as *s.calculateArea()*, the runtime will determine which method to be executed based on the actual pointed object in runtime.
- Therefore the same method call above may result in different responses varying from different underlying objects.
- Polymorphism allows you to write code to deal with a group of similar objects with common parent. As a result, your program will be more generic and clean.

PolymorphismDemo

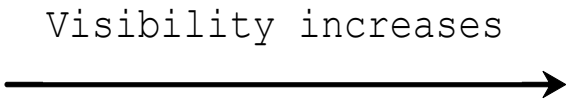
The instanceof Operator

- Use the instanceof operator to test whether an object is an instance of a class:

```
Object o1 = new Circle();
Object o2 = new Triangle();
testObject(o1);
testObject(o2);

void testObject(Object obj) {
  if (obj instanceof Circle)
    System.out.println("The object is Circle");
  else if (obj instanceof Triangle)
     System.out.println("The object is Triangle");
  else
     System.out.println("Neither of them");;
}
```

The protected Modifier

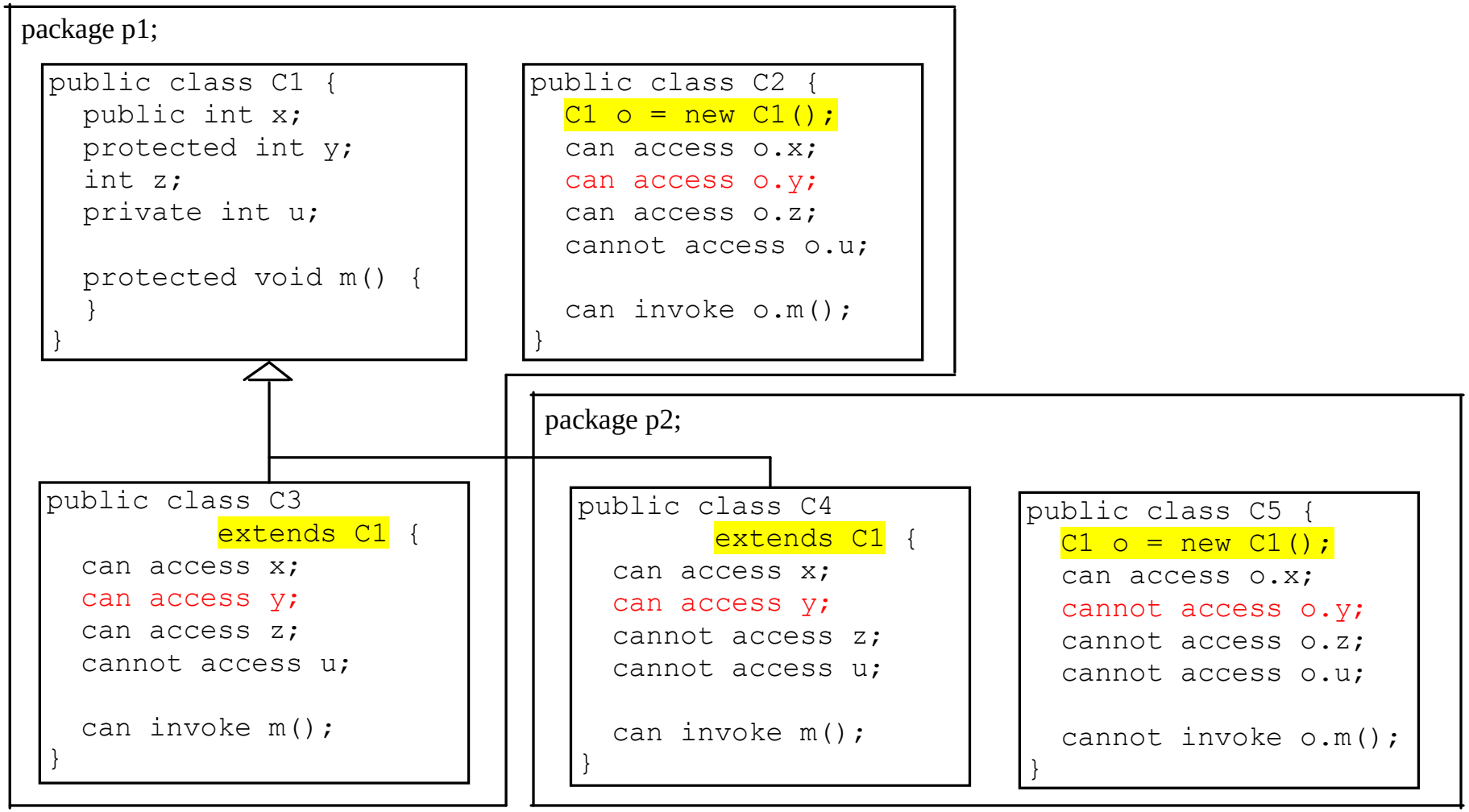
- The protected modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.



`private, none (if no modifier is used), protected, public`

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	-
default	✓	✓	-	-
private	✓	-	-	-

Visibility Modifiers



Restriction on Overriding

- A subclass cannot weaken the accessibility of a method defined in the superclass.
- For example, if a method is defined as public in the superclass, it must be defined as public (not private, package or protected) in the subclass.
- A subclass may override a protected method in its superclass and change its visibility to public only (or leave it unchanged as protected).

The final Modifier

- The keyword final can be used to declare a constant. It can also be applied to a class and method.
  - The final variable cannot be changed (constant):

    ```
    final static double PI = 3.14159;
    ```

  - A final class cannot be extended:

    ```
    final class CannotExtendFromMe {
       ...
    }
    ```

  - The final method cannot be overridden by its subclasses.

Useful APIs

Scanner

Sort Array

Search Array

ArrayList

Scanner

- You can get input from console using Scanner object:

1. Create a Scanner object (the counterpart of scanf() in C language)
   - Scanner scanner = new Scanner(System.in);

2. Use these methods next(), nextByte(), nextShort(), nextInt(), nextLong(), nextFloat(), nextDouble(), or nextBoolean() to obtain to a string, byte, short, int, long, float, double, or boolean value. For example,
   - System.out.print("Enter a double value: ");
   - Scanner scanner = new Scanner(System.in);
   - double d = scanner.nextDouble();

Searching Arrays

- Java provides several overloaded binarySearch methods for searching a key in an array of int, double, char, short, long, and float in the java.util.Arrays class. For example, the following code searches the keys in an array of numbers and an array of characters.

- If an element exists:

```
int[] list = {2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79};
System.out.println("Index is " + java.util.Arrays.binarySearch(list, 11));   // Return is 4
```

- If an element does not exist, a negative value will be returned.
- The array must be pre-sorted in increasing order, otherwise the results are undefined.
- If the array contains multiple elements with the specified value, there is no guarantee which one will be found.

Sorting Arrays

- Sorting, like searching, is also a common task in computer programming.

- Since sorting is frequently used in programming, Java provides several overloaded sort methods for sorting an array of int, double, char, short, long, and float in the java.util.Arrays class. For example, the following code sorts an array of numbers and an array of characters.

  - double[] numbers = {6.0, 4.4, 1.9, 2.9, 3.4, 3.5};
  - java.util.Arrays.sort(numbers); // no return, sort in place

  - char[] chars = {'a', 'A', '4', 'F', 'D', 'P'};
  - java.util.Arrays.sort(chars); // no return, sort in place

ArrayList

- An array's size is fixed once the array is created.
- Java provides the ArrayList class that can be used to store an unlimited number of objects. (i.e. scalable array)

java.util.ArrayList	
+ArrayList()	Creates an empty list.
+add(o: Object) : void	Appends a new element o at the end of this list.
+add(index: int, o: Object) : void	Adds a new element o at the specified index in this list.
+clear(): void	Removes all the elements from this list.
+contains(o: Object): boolean	Returns true if this list contains the element o.
+get(index: int) : Object	Returns the element from this list at the specified index.
+indexOf(o: Object) : int	Returns the index of the first matching element in this list.
+isEmpty(): boolean	Returns true if this list contains no elements.
+lastIndexOf(o: Object) : int	Returns the index of the last matching element in this list.
+remove(o: Object): boolean	Removes the element o from this list.
+size(): int	Returns the number of elements in this list.
+remove(index: int) : Object	Removes the element at the specified index.
+set(index: int, o: Object) : Object	Sets the element at the specified index.