Java

- The term Java is not an acronym
  but adopted to reflect a favorite drink (coffee?) of many programmers – hence Sun's logo for Java is a cup of steaming coffee.
- http://java.sun.com
- JDK = Java Development Kit
- Java is a case-sensitive language
- Java code can be written as either an application or an applet
- API = Application Programming Interface
- The ***java.lang*** package is automatically imported.
- abstraction denotes the essential properties and behaviors of an object that differentiate it from other objects
- **Object-oriented Program Design** → modeling abstractions using classes and objects
- UML : Unified Modeling Language notation
- Objects are manipulated through object references (reference values / references)
- infix dot '.' operator
- everything must be encapsulated in classes
- 2 kind of values
  - atomic values of primitive types
  - reference values

Program Implementation

- program keyed in from document ⟶Editor→ program stored on a disk in text mode ***source code*** (.java)
- program stored on disk in text mode⟶Compiler→program stored on disk as a series of bytecodes (.class)
  - **Java byte codes** ⇒ a set of instructions written for a hypothetical computer, known as **Java virtual machine**
  - Regardless of the computer you are using, the compiler will generate the same Java byte code program ⇒ programs written in Java are **portable**.
  - on JDK: javac ClassName.java
  - one .class file for each class/interface
- program as series of bytecode is stored in memory ⟶Interpreter(←data)→ results
  - on JDK: java name → invoking the main() method from the specified class
  - java : Java interpreter
- **block** → many statements contained between braces { }
  - If only one statement is executed in a selection statement, the use of braces can improve the clarity of the code, even though the braces are themselves redundant.

Template for constructing a Java application program

```
// heading giving details of the name and purpose of the program
import java.io.*;
// java.io.BufferedReader, java.io.PrintWriter
class ClassName
{
    // declarations of input and out streams
    static  BufferedReader keyboard = new
        BufferedReader(new InputStreamReader(System.in));
    static  PrintWriter screen = new PrintWriter(System.out, true);

        // main method
    public static void main (String[] args) throws IOException
        // readLine() in the class BufferedReader throws an
        IOException
        // args can be any name e.g. arg, etc.
    {
        // declaration of constants

        // declaration of variables

        // program statements

                screen.print("Input something "); screen.flush();
        input = new Float(keyboard.Leadline()).floatValue();
        screen.println("output = " + input);
    }
}
```

- the order of the static and public keywords is irrelevant

Object

- has a unique identify
- has its own copy of the variables declared in the class definition
- do not have names but are denoted by references
- can only be manipulated via references, which can be stored in variables
- can have several references → **alias**es
- deletion is taken care of by the runtime system
- state: the values of variables inside
- communicate by message passing
- cannot contain other objects; can only have references to other objects

Statements terminated by a semicolon

- ; → empty statement
- {} → compound statement

Assignment Statement

```
identifier = literal;
identifier = identifier;
identifier = expression;
```

- destination is always on the left-hand side
- The assignment of one object to another of the same type does not create a copy of the object

note

- 9=a → has no meaning, since 9 is not a legal identifier

- a=a+1 → increase the value of the variable *a* by 1

Legal assignment? (compile-wise)

- At compile time, refA = refB will be checked by the type of the refs not the actual type of the object that they refer to.
- On the hierarchy diagram, if can follow the arrow from class/interface B up to class/interface A, then it's valid
- refOfSuper = refOfSub is valid
- // refOfSub = refOfSuper is not valid : compile error (even when the refOfSuper really refers to the sub object)
- refOfSub = (Sub) refOfSuper is valid
    - may cause ClassCastException at run-time if
        - refOfSuper actually refers to a super object
        - refOfSuper refers to object of another subclass of that super class

The rules for reference assignment (enforced at compile time)

```
SourceType srcRef;
DestinationType destRef = srcRef;
```

- If SourceType is a class type, valid if
    - DestinationType is a superclass of the subclass SourceType
    - DestinationType is an interface type which is implemented by the class SourceType
- If SourceType is an interface type, valid if
    - DestinationType is Object
    - DestinationType is a superinterface of subinterface SourceType
- If SourceType is an array type, valid if
    - DestinationType is Object
    - DestinationType is an array type, where the element type of SourceType can be converted to the element type of DestinationType

Type conversion

- for primitive datatypes
  widening conversions are permitted
  narrowing conversions require an explicit cast
- for reference values
    - upcasting: conversions up the inheritance hierarchy are permitted
    - downcasting: conversions down the hierarchy require explicit casting
- performed automatically when
  the type of the expression on the right-hand side of an assignment can be ***safely*** promoted to the type of the variable on the left-hand side.

Ex not allow : int=long

cast

```
(DestinationType) ref;
```

Ex allow: int = (int)long

- digits may be lost in the assignment
- cast conversions are unsafe, as they may throw a ClassCastException at runtime

Legal cast? (compile-wise)

- (super) subRef is valid
- (sub) superRef is valid
    - but if the superRef actually refers to a super object, will cause runtime error: throws ClassCastException

Ex

```
public class Class1
{
    public static void main (String[] args)
    {
        A a; B b; C c; D d;
        a = new A();
        b = new B();
        // b = (B) a; //not OK, will throw a
java.lang.ClassCastException when run
        a = b; //a ref to subclass B object
        // b = a; // not OK, compile error, can't implicitly convert
A to B
        //c = (C) a; //not OK, compile error, can't convert A to C
        b = (B) a;
        a = new A();
        // b = (B) a; //not OK, will throw a
java.lang.ClassCastException when run
        d = new D();
        a = d;
        // b = (B) a; //not OK, will throw a
java.lang.ClassCastException when run
    }
}
class C{}
class A{}
class B extends A{}
class D extends A{}
```

instanceof

```
ref instanceof DestinationType
```

- true : if ref can be cast to the DestinationType (class, interface, array type)
  ≡ the corresponding cast expression is valid
- false : the cast involving the operands will throw a ClassCastException
    - literal "null" is not an instance of any reference type
    - any array of non-primitive type is an instance of both Object and Object[] types
    - an instance of a superclass is not an instance of its subclass
    - an instance of a class cannot be of an interface type which is not implemented by the class of the object
    - an instance of a class is not an instance of a totally unrelated class
        - This can pass the compile check by reference one of then via an Object reference
- what matters is the class of the actual object denoted by the reference at runtime, not the type of the reference
-

compile time check:

refDestinationType = (DestinationType)refSourceType
refSourceType instanceof DestinationType
whether a refSourceType and a reference of DestinationType can denote objects of a class (or its subclasses) where this class is a common subtype of both SourceType and DestinationType.
If this is not the case, then obviously there is no relationship between the types, and neither the cast nor the instanceof operator application would be valid
Therefore, if no common subtype

⇒ compiler will reject casting refSourceType to type DestinationType or applying the instanceof operator

Class

- The name of the class containing the ***main*** method must be the same as the the name given to the program file (omitting the .java suffix).
- convention: the name of a class should always begin with an upper-case letter.

(modifier) class ClassName <extends …> <implements …,…,…>{}

- may contain
  - data → properties/attributes → variables/fields
    - constants
    - class data: *static*
      - when class is loaded, static variables are initialized to their default values if no other explicit initialization is provided
      - initialized when the class is loaded at runtime
    - instance variables ⇒ represent the data for a particular object
      - Each object will have its own set of instance variables, which represent the state of an object
  - methods → behaviors → operation
    - constructors
    - class methods: static
    - instance methods
  - or both
- access static members
  - ClassName.memberName
  - ClassObjectRef.memberName
- access instance members → ClassObjectRef.memberName only

static member

- can access these by using the class name, or through object references of the class
- the class need not be instantated to access its static members
- not instantiated when an instance of the class is created

Method

⇒ a group of self-contained declarations and executable statements that perform an activity.

⇒ a group of declarations and executable program statements that perform a particular activity

<modifier> returnType methodName(formal-parameter-list)<throws…,…,…>;
  - uniquely identify a method in terms of its return type, name and formal parameter list.

- may represent activities associated with a particular classification or data type
- **signature** → name, types, numbers, order of parameters
- If no data is returned to the caller, the keyword ***void*** is used for the return-type.
- formal parameter list → a comma-separated list of parameters

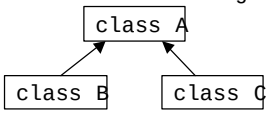
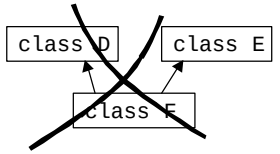
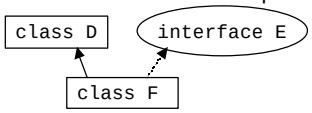
Calling/invoking

- A class method is invoked by a direct call to the method
- an instance method is invoked by an object of the same class
- When calling a method,
  actual-parameter list : the list of literals or variables, enclosed in parenthesis after the method name
- Both the and **the formal-parameter list** must
  - contain the same number of arguments,
  - in the same order
  - of the same data type
- The names of the identifiers in the actual parameter list and the formal parameter list can be the same or different.
- The computer will return to the calling method by either executing a return statement or by reaching the physical end of the method

passing arguments

- parameters are passed by value
- values of actual parameters must be assignable to formal parameters of compatible types.

Value Parameters	**Reference Parameters**
• primitive data type	• object and array (stored by reference)
• evaluate the argument and create a local copy of the value, assigning it to the corresponding parameter in the called method	• the references to the object or array is passed and not the specific values of the object or array.
• any change to the parameters would be localized to the function and would not change the values in the *main* method	• any changes made to the values of the values of the parameters in the called method will result in changes being made to the values of the corresponding arguments in the calling method.

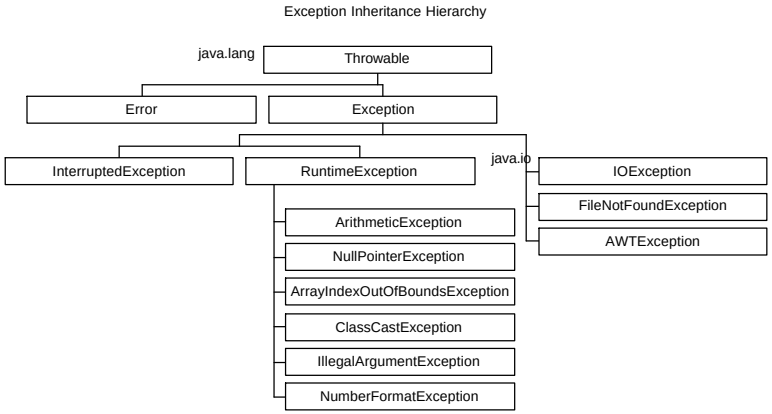
- ```
  public static void main(String[] args)
  {
      int a = 2;
      change(a);  //a = 2, still
      int[] b = {1, 1, 1};
      change(b);  //now, b[1] = 5
  }
  ```
  ```

```
public static void change(int a)
{
    a = 5;  //= 5 only inside this method
}
public static void change(int[] a)
{
    a[1] = 5;
}
```

The return Keyword

- `return expression;`
- expression may be omitted
- dual purpose
  - assigns a value to the method
  - marks the position in the method where the computer must return to the calling method
- *void* :
  - may use a *return* statement without an expression to force the computer to return to the caller
  - if *return* statement is omitted, the computer will automatically return at the end of the method
- may contain several *return* statements when there are places in a class method that logically allow for the termination of the execution of the method.

Constructors

- same name as the class
- can not return a value
- can only be called using the "new" operator
- A class containing a constructor and instance methods, may be thought of as a data type
- A variable declared as a class type does not become an object until a constructor within the class has been executed.
- is normally used in conjunction with the keyword *new* which allocates memory space from the heap.
- provides the storage in memory and the initialization of the instance variables allocated to the object.
- For each separate invocation of the constructor, a new object will become instantiated.
- default constructor → no parameter
- if a class does not specify any constructors, implicit default constructor is supplied: ClassName(){} //No parameters. Empty constructor body
- Caution: if provide non-default constructor, no-arg constructor will not be automatically created.

Instantiation

= allocation of memory for storing the object's data and the initialization of this memory space with appropriate values
= creating an **instance** of the class
= creating an **object**

- Instantiation is made possible by the use of a constructor

```
ClassName objectName = new ClassConstructor();
ClassName objectName = new
                       ClassConstructor(argumentList);
```

- **new** : allocate a new memory storage area for holding the value of the object
  - primitive data is stored by **value**
    - can be conceptually represented in the memory of the computer
  - object is stored by **reference**
- simply say "Object o" does create any object. Have to say "new Object()" to actually create an object.
- when an identifier is initialized, the value of the object is not stored at the memory location depicted by the identifier, but stored in a different location pointed at or *referenced* by the identifier
- **default**:
  - Instance variables and static variables receive a default value unless explicitly initialized
  - Local variables remain uninitialized unless explicitly initialized.

Class method

- **static** : cannot be invoked by an object
- invoked by using the name of the method
- can directly access other static members in the class
- cannot access instance members of the class (there is no object being operated on when a static method is invoked)
  - can always use a reference of the class's type to access its members, regardless of whether these members are static or not
- A Java application consists of at least one class method, the *main* method.
  - signature: `public static void main (String[] args);`
  - is executed before any other method ⇒ The computer will start the execution of the program at the first statement in the *main* method, and terminate execution after the last statement.
  - There must be one *main* method present in only one of the classes.
  - When any predefined method throws an exception, append a throws clause to the first line of the *main*, listing the name of the exception(s).

Instance method

a group of methods that appear to describe the characteristics and operations you might associate with an object

- belong together in the context of the description of the class to describe the state and behavior of an object.
- are used to perform a variety of operations that pertain to the object.
- may have modifiers, a return type, a name and a formal parameter list
- Invoking an instance method by object (not called directly, as with class methods)

```
object.methodName();
object.methodName(argumentList);
```

- can only be invoked on objects of the class

- its body **can access all members defined in the class**
- are passed an implicit parameter which is a reference to the object on which the method is being invoked
  This object can be referenced in the method's body by the keyword "this"

This

- this can only be used in non-static method
- cannot be modified
- refers to the current object
  = object being instantiated for constructor
  = object invoked the call for instance method
  - For constructor, the *this* object is implicitly returned
  - Don't have to use except there is a local variable with the same name as instance variable ⇒ use the name for local variable, and this.name for instance variable
    (use ClassName.name for static variable)
- can use inside constructor to call peer overloading constructor.
  - must be first line in constructor

Method Overloading

- using the same name for methods: constructors / instance methods / class methods (but not a mixture of all three)
- changing just the return type or the exceptions thrown is not enough to overload a method → compile error
- the number and type of parameters in the formal parameter list is the only in which the compiler can distinguish overloaded methods
- **no** operator overloading
  - **Exception** : overloading of the + operator for string concatenation

Inheritance and Class Hierarchy

⇒ the process by which one class receives the characteristics of another class

class *SubclassName* **extends** *SuperclassName*

- **superclass** → initial class, base class, parent class
- **subclass** → receiving class, derived class, child class
- hierarchy diagram : arrows always point from subclass to superclass.
- an object of any subclass in the hierarchy is also a legal superclass object
  - An object of a subclass may be assigned to an object of its superclass without a data type violation
  - An object of a subclass may be passed as an argument to a method that requires a parameter of it superclass type
- private variables are inherited by subclass objects
  - each such object has its own copy of the variable with its own value
  - cannot be accessed directly by the object itself
  - can only be accessed through any protected or publiced or public access method of the superclass
- for class variable, can redefine so it get its own independent value
- subRef.superMethod() is valid
- the only superclass Reference that can be assigned to a subclass-typed variable, even with an appropriate cast, are those superclass ref that are actually subclass objects
- several classes may inherit from a single class



- a single class cannot inherit from more than one class



- All Java classes are ultimately derived from the Object class
  - it is not strictly necessary to state that a subclass extends the Object class
- subclass is not inherited the constructor(s) from superclass

super

Constructor

- if present in a constructor, must always be the first statement in a constructor body
- super() refers to the default, no-argument constructor, of the superclass
- may refer to parameterized constructors of the superclass
- **When you construct an object of a subclass, the constructor for the superclass also get invoked.**
  - if omitted in subclass constructor, Java will automatically insert **super()**
    - If the superclass does not contain a default (no argument) constructor, this will result in a compilation error.
  - **Caution**: Compile error if superclass doesn't have no-arg constructor (doesn't defined no-arg and define any other constructor → no automatically-created default constructor)
- Constructor calls are automatically chained.
  - A sequence of constructor methods are invoked from subclass to superclass and eventually to the Object class
  - a superclass constructor is always invoked before the subclass constructor
  - the body of the Object constructor is executed first, followed by the execution of the bodies of the constructors down through the class hierarchy, and finally to the execution of the subclass constructor body

variable

- may be used as a prefix to access inherited variables and inherited methods of a superclass in a subclass

Overriding Superclass Method

- the superclass is **overridden**, when a subclass defines a method with the same name, return type and argument list as a method in a superclass
- must be at least as accessible as they are in superclass
- method definition in the subclass can only specify **all or subset** of the exception classes (including their subclasses) specified in the throws clause of the overridden method in the superclass
- the inherited method can be accessed by the statement `super.methodName()`.

Dynamic Method Lookup

⇒ the process of determining which method definition a method signature denotes during runtime, based on the class of the object

⇒ a technique where each object has a table of its methods, and Java searches for the correct versions of any overridden methods at run-time

- When a method is invoked using a reference, the method definition which actually gets executed is **determined both by the class of the actual object** denoted by the reference at runtime and the method signature
- AnObjRef.method() will cause compile error if AnObjRef is a reference of a class/interface that doesn't have method() even when AnObjRef denotes the actual object that is one of the class that have method().
- the compiler doesn't know; the decision on which method to use is poseponed until runtime
- Dynamic method is not as fast as invoking a method directly.
- Dynamic method lookup is not required for static or private methods and those methods and classes declared as **final**.

polymorphism

⇒ Java'a ability to decide amongst methods based on the runtime class

⇒ a way of giving a method one name that is shared up and down an object hierarchy, with each object in the hierarchy implementing the method in a way appropriate to itself

- To write polymorphic classes
  - The classes must be part of the same inheritance hierarchy
  - The classes must support the same set of required methods

Shadowed Variables

- The variable of the subclass **shadow** the inherited variable, if an inherited variable has the same name as a variable of the subclass
- to use the inherited variable in the subclass, use the reserved word *super*
- If class ChildClass is a subclass of class ParentClass, and both contain a variable named common
  In class ChildClass,
  - the variable common may be refered to by
    - common
    - this.common
  - the inherited variable common is refered to by
    - super.common
    - ((ParentClass)this).common

  Outside both class
  - may refer to shadowed variables by casting an object to the appropriate type
- Variable is **determined by the reference type**, not the actual type of the object

Is-a and has-a Relationship

📖 In the is-a hierarchy, we can say that inheritance is appropriate if every object of class Y may also be viewed as an object of class X

📖 The has-a relationship describes that every object of a class X has-a set of attributes of type Y.

Encapsulation

⇒ an approach to program development that attempts to hide much of the implementation details of a class.

- the interface of each class is defined in such a way as to reveal as little as possible about its inner workings
- access to the data is allowed only via specific instance methods
- the implementation of the data, constructors, instance methods, and class methods may be hidden from the user

⇒ the grouping together of data and a set of methods to perform actions on the dat

ADT: abstract data type

- en encapsulated group, consisting of a data and its associated methods

Abstract class

⇒ a class that contains at least one abstract method

- not all the methods of an abstract class need be abstract.
- at the top of the hierarchy
- An object cannot be instantiated
- an object may be declared as being of abstract type
- acts as a blueprint for all subclasses
- can be extended to suit different classes within the taxonomy
- A subclass of an abstract class may be instantiated, provided that all abstract methods of the abstract class are overridden and implemented in the subclass.
- If not all the abstract methods are implemented, the subclass must also remain abstract

abstract method

- difined by the method's signature
- has no method body
- methodName();

Interfaces

⇒ a class that must contain only abstract methods and/or constant

- references of interface type can be declared
  these can denote objects of classes that implement this interface
  (upcasting)
- possible for an interface not to contain any methods or constants
- Interface are a data type, in the same way that classes are a

data type.

- When a class implements an interface, instances of that class can be assigned to variables of the interface type.
- Multiple inheritance of interfaces is allowed in Java

- class D{…}
  interface E{…}
  class F extends D implements E{…}



Commond Line Arguments

- The arguments that are passed (parameter list for the *main* method)
  - are stored as strings in the array *args*.
  - *args[0]*, *arg[1]*, and so on
- The programming environment that you are using → will dictate the manner in which you are allowed to pass arguments to the *main* method.
- asun or equivalent Solaris-based computer:
  enter the parameters after the name of the executable program
- fully integrated program development environment, such as Microsoft's Visual J++™:
  enter the parameters in a space window before executing the program
  open project settings and, in the stand-alone interpreter mode, input the program arguments.

Scope of Identifiers

- the region of a program in which an identifier can be used

class scope

- accessible from its point of declaration throughout the entire class.

block scope

- only accessible from the point of declaration to the end of the block.
- But can passed as an argument
- is not always recognized by the use of braces { }
  - any variable defined by the *for* loop is visible in the statement that follows the for definition.
- identifiers of a formal parameter list may also be regarded as having block scope

Lifetime of Identifiers

⇒ the period during which the value of the identifier exists in computer memory.

- Identifiers declared as being *static* exist for the life of the program
- parameters and identifiers having block scope only exist during the execution of the method
- When an object goes out of scope, the amount of memory allocated to storing that object is returned back to the heap for future use by other objects.
- ***heap*** → an area of memory set aside for the dynamic allocation of computer memory to objects during run time.
- ***garbage collection*** → the java system automatically returns memory to the heep when it is no longer required

Java access specifiers

- Each access specifier controls the access for only that particular definition.
  - *C++ ® the access specifier controls all the definitions following it until another access specifier comes along.*

To grant access to a member

- Make the member **public**. Then everybody, everywhere, can access it.
- Make the member friendly by leaving off any access specifier, and put the other classes in the same package. Then the other classes can access the member.
- Inherited class can access a **protected** member as well as a **public** member (but not **private** members). It can access friendly members only if the two classes are in the same package.
- Provide "accessor/mutator" methods (also known as "get/set" methods) that read and change the value.

Member Accessibility	private	package	protected	public
From a subclass in the same package	yes	yes	yes	yes
From a non-subclass in the same package	no	yes	yes	yes
From a subclass in another package	no	no	yes	yes
From a non-subclass in another package	no	no	no	yes

private

Member → visible within its own class

- no one can access except that particular class, inside methods of that class
- outside class
  - cannot be accesses directly by the object itself
  - can only be accessed through any protected or public access method
- allows you to freely change that member without concern that it will affect another class in the same package.
- members are still inherited, but they are not accessible in the subclass
- private constructor
  - can't use `new ClassName()` outside its own class → may have to provide a public method inside

```
static ClassName makeAnObject()
{
    return new ClassName();
}
```

    - If you don't want anyone else to have access to that class, you can make all the constructors **private**, thereby preventing anyone but you, inside a **static**

member of the class, from creating an object of that class

- //option(1): Allow creation via static method

```
class Soup
{
    private Soup() {}
    public static Soup makeSoup()
    {
        return new Soup();
    }
    public void f(){}
}
//outside this class:
Soup s = Soup.makeSoup();
s.f();
```

- //option(2): Create a static object and return a reference upon request

```
class Soup
{
    private Soup() {}
    private static Soup ps1 = new Soup();
    public static Soup access()
    {
        return ps1;
    }
    public void f(){}
}
//outside this class:
Soup s = Soup.access();
s.f();
// a "singleton" pattern ⇒ allows only a single object to ever be created and you can't get at it except through the public method access( ).
```

  - If the default constructor is the only one defined, and it's **private**, it will prevent inheritance of this class

- Any method that you're certain is only a "helper" method for that class can be made **private,** to ensure that you don't accidentally use it elsewhere in the package and thus prohibit yourself from changing or removing the method. Making a method **private** guarantees that you retain this option.
- Unless you must expose the underlying implementation (which is a much rarer situation than you might think), you should make all fields **private**.

class may not be private

default access / Friendly / package

- no access specifier at all
- has no keyword

member

- all the other classes in the current package have access to the friendly member
- to all the classes outside of this package the member appears to be private

class

- can be used only within that package.
- an object can be created by any other class in the package, but not outside the package
- Since a compilation unit—a file—can belong only to a single package,
  all the classes within a single compilation unit are automatically friendly with each other.
  Thus, friendly elements are also said to have *package access*.
- allows you to group related classes together in a package so that they can easily interact with each other
- all source files that are in the same directory and have no explicit package name, are treated by Java as implicitly part of the "default package" for that directory, and therefore friendly to all the other files in that directory.
- If you create a new package and you inherit from a class in another package, the only members you have access to are the **public** members of the original package. (Of course, if you perform the inheritance in the *same* package, you have the normal package access to all the "friendly" members.)

protected

- variable → can be accessed from any method of any class in the same package
- still has "friendly" access within package
- a subclass in another package can only access protected members in the superclass via references of its own type or a subtypes.

public

- variable and method → visible anywhere its class is visible
  - always use *public static void main(String args[])*
- class → visible anywhere
  - Compile error:
    - There can be only one **public** class per compilation unit (file).
      - each compilation unit has a single public interface represented by that **public** class
      - can have as many supporting "friendly" classes as you want
    - The name of the **public** class must exactly match the name of the file containing the compilation unit, including capitalization.
    - It is possible, though not typical, to have a compilation unit with no **public** class at all. In this case, you can name the file whatever you like.

Note

- Just because a reference to an object is **private** inside a class doesn't mean that some other object can't have a **public** reference to the same object.
- you have only two choices for class access: "friendly" or **public**.
- an inherited class
  - can access a **protected** member as well as a **public** member (but not **private** members).
  - can access friendly members only if the two classes are in the same package

Top-level Nested Classes and Interfaces

- also consider to be at the top level like an ordinary class or

interface
- defined as a static member of an enclosing top-level class or interface
- can be nested to any depth, but only within other static top-level classes and interfaces
- Interface are implicitly static. Nested interface can optionally be prefixed with the keyword static and have public accessibility
- cannot have the same name as an enclosing class or package

Data Types

- **Hexadecimal numbers** are prefixed by 0x
- **Primitive data types** = char, int, long, float, double
- **Unicode** ⇒ prefixed by \u
- **literal** ⇒ the stated value
  - character : always delimited by single quotes
- A = \u0041

		postfix	#bit	min	max
char	Character		16 Unicode		
int	Integer		32	-2^{31}	$+2^{31}-1$
long	Integer	l or L	64	-2^{63}	$+2^{63}-1$
float	real	f or F	32	$\pm1.4x10^{-45}$ (1.40239846)	$\pm3.4x10^{38}$ (3.40282347)
double	real	d or D	64	$\pm4.94x10^{-324}$ (4.94065645841246544)	$\pm1.8x10^{308}$ (1.7976931 3486231570)

- The use of a plus sign (+) is optional for positive literals.
- All decimal integer literals must begin with a digit in the range 1…9 after the sign if one is present.
- Integer literals must not begin with 0 (zero)
- A real literal can be written in one of two ways: –123.456 or –1.23456E+2
- double : double-precision
- default
  - real : double
- \t → for a tabulation
  \n → for a new line
- **Boolean Data Type**
  - permitted to have only one of two values → true or false
  - initialized by Java to be *false*
- A ***white space*** character is generally regarded as either a
  - space '\u0020'
  - horizontal tabulation '\u0009'
  - new line '\u000A'
  - vertical tabulation '\u000B'
  - form feed character '\u000C'
- A ***wildcard*** is a character that can represent a number of different character.
  - The wildcard * may represent any of the class names.
- the \uxxxx notation can be used anywhere in the source to represent Unicode characters

Identifiers

- may contain
  - A-Z, a-z
  - $
  - _ (underscore character)
  - 0-9
- may start with any characters with the *exception* of a decimal digit
  - may but avoid starting an identifier with _ or $
    - Often such characters are used in other variables by the computer
- can normally be of any practicable length
- must not be the same as Java keywords
- When an identifier is constructed from more than one word, each successive word should begin with an upper-case letter

KEYWORD

- Thoses words in a non-bold regular typeface are reserved by java but currently unused.
  abstract, boolean, break, byte, byvalue, **case,** cast, **catch, char, class,** const, **continue, default, do, double, else, extends, false, final, finally, float, for,** future, generic, goto, **if, implements, import,** inner, **instanceof, int, interface, long, native, new, null,** operator, outer, **package, private, protected, public,** rest, **return, short, static, super, switch, synchronized, this, throw, throws, transient, true, try,** var, **void, volatile, while**

DECLARATION AND INITIALIZATION

- data declarations must appear before the data instructions

Variables

- Variable Declaration:
  dataType identifier;
  dataType identifier1, identifier2, identifier3;
  - Java automatically initializes all data of a primitive type at the point of declaration.
    - interger → 0
    - real → 0.0
    - characters → \u0000
- Variable Initialization:
  dataType identifier = literal;
- Only primitive data and reference values can be stored in variables

Constants

- must be initialized at the point of declaration
- final dataType IDENTIFIER = literal;
- Convention: constant identifiers should be code in *upper-case letters*

Unary operators ➧ have one operand
Binary operators ➧ have two operands

Referance

- Polymorphism allows a reference to denote different objects in

the inheritance hierarchy at different times during execution

- a superclass reference can denote objects of its subclasses
- an interface reference can denote objects of classes that implement the interface

Operator Precedence

priority level	type	operator	symbol
1	unary	negate	-
1	unary	plus	+
1	cast	a data type	(type)
2	multiplicative	multiply	*
2	multiplicative	divide	/
2	multiplicative	remainder	%
3	additive	add	+
3	additive	substract	-
13	assignment	equal	=

- Expressions are evaluated by taking the operators with a higher priority before those of a lower priority.
- Generally, where operators are of the same priority, the expression is evaluated from left to right.
- Expressions in parenthesis will be evaluated before non parenthesized expressions.
- Parenthesis, although not an operator, can be considered as having an order of precedence after unary operators.

Increment/ Decrement Operators

- counter = counter + 1 → counter++
- counter = counter -1 → counter–
- System.out.println(counter++) will print the old counter not the new one

Conditional Expressions

- can equate to one of the two values, either true or false
- ***short-circuit evaluation***
  - o only the condition X need be evaluated.
  - o when using logical AND, if condition X is false
  - o when using logical OR, if codition X is true
  - o Both logical && and ||use short-circuit evaluation.
  - o To avoid short-circuit evaluation, use the corresponding logical operators **&** and **|**.

SELECTION

Rational operator

operator	meaning
>	greater than
<	less than
==	equal to
>=	greater than or equal to
<=	less than or equal to
!=	not equal to

- Comparison of real numbers for equality should be avoid, since real numbers are not always accurately stored by the computer.
- use the instance methods *compareTo, equals* and *equalsIgnoreCase* defined in the *String* class, if need to compare the value of strings

If…else

- if (conditionalExpression){statementWhenTrue;} else {statementWhenFalse}
- if (conditionalExpression){statementWhenTrue;}
-
```
if( )
{
    if( )
        …;
    else
        …;
}
else
{
    if( )
        …;
    else
        …;
}
```
-
```
If ( )
     if ( )
          …;
     else
          …;
```
-
```
if ( )
{
     if ( )
          ;
}
else
     ;
```
-
```
if (reply.equals("YES"))
{
    ;
}
else
{
    if (reply.equals("NO"))
    {
        ;
    }
    else
        error = true;
}

if (error)
    screen.println("DATA ERROR - reply not in correct format");
else
    ;
```

Else If

```
if  ( )
    ;
else if ( )
    ;
else if ( )
    ;
else if ( )
    ;
else    ( )
    ;
```

Switch

```
switch (expression)
{
    case c1: statement(s); break;
    case c2: statement(s); break;
        .
        .
    default: statement(s); //optional
}
```

- for selection based upon an ordinal type
- The expression must evaluate to an ordinal value
- Those values that are not represented by case labels will result in the statement after the optional default being executed.
- If the optional ***default*** statement was not present and the value of expression does not satisfy any case, then the computer would branch to the end of the ***switch*** statement.
- Without default option, no action would occur when a value was out of range.
- necessary to include a way of exciting from the ***switch*** statement at the end of every ***case***.
    - Failure to exit from the ***switch*** will result in the execution of all the case statements following the chosen case.
    - One method of exiting from a switch statement is through the use of a **break** statement at the end of every ***case*** list. → causes the ***switch*** to terminate, and execution resumes with the next statement (if any) following the end of the ***switch*** statement.
- switch ( )
  {
  case 1: case 3: case 5:
  ; break;
  case 2:
  ; break;
  }
- ***An ordinal variable*** → has a value that belongs to an ordered set of items , eg
    - integer
    - character
- Real numbers and strings are not ordinal types.

REPETITION

While

```
while (conditional-expression)
    {statement(s);}
```

Do¼ While

```
do statement(s) while (conditional-expression);
```

- always permits the statements within the loop to be executed at least once by the computer.

For Loop

```
for (expression1; expression2; expression3) statement(s)
```

- expression1 → the declaration (if necessary) and initialization of the loop control variable
    - If expression1 is omitted, → the initialization (and declaration) of the loop control variable must take place before entry in the loop
- expression2 → a condition under which repetition will continue
    - If expression2 is omitted, → then the loop does not terminate unless it contains a ***break*** statement.
- expression3 → statement to increment or decrement the loop variable.
    - If expression3 is omitted, → then increasing or decreasing the loop variable must take place within the body of the loop.
- The expressions in a ***for*** loop are optional
- Even when the expressions are omitted in for loops, the semicolon separators must be present
    - infinite loop

```
for( ; ; )
{
    screen.println("forever and ever…");
}
```

- for (expression1; expression2; expression3) statements(s);
- expression1;
  while (expression2)
  {
  statements(s);
  expression3;
  }
- is normally used for counting.
- is a specialization of a ***while*** loop.
    - can always replace a ***while*** loop with a ***for*** loop.
    - cannot always replace a ***for*** loop with a ***while*** loop

Inner Classes

- There are no non-static inner, local or anonymous interface

Non-static Inner Classes

- defined without the keyword static, as members of an enclosing class
- can be nested to any depth
- on par with other non-static members defined in a class
- an instance
    - can only exist with an instance of its enclosing class.
    - must be created in the context of an instance of the enclosing class.

Ex Outside the enclosing class use **enclosingObjectRef.new Inner()**

```
Outer outerRef = new Outer();
Outer.Inner innerRef = outerRef.new inner();
```

or

```
Outer.Inner innerRef = new Outer().new Inner();
```

or

```
Outer.Inner.Inner innerOfInnerRef = new Outer().new
Inner().new innerOfInnerRef();
```

- // new Outer.Inner() → compile error

Ex In the enclosing class's non-static member

- The return type can be Inner or Outer.Inner
- can say `new Inner();`
- instance variable: `Inner innerRef = new Inner();`

Ex In the enclosing class's static member

- static variable:
  `static Inner innerRef = new Outer().new Inner();`
- local variable in static method :
  `public static void method(){Inner innerRef = new Outer().new Inner();}`

- **cannot** have static members
  (the class does not provide any services, only instances of the class do)
- can have non-default constructors
- instance methods can directly refer to any member (including class) of any enclosing class, including private members. No explicit reference is required.
  - Explicit reference to members in the enclosing class : `Outer.this.name` or `Outer.this.name()`
- **Shadowed Members**
  In InnerOfInner class's non-static method ⇒ can have
  - Name , this.Name , InnerOfInner.this.name
    → denote one of InnerOfInner
  - Inner.this.name → denote one of Inner
  - Outer.this.name → denote one of Outer
  - These members are not overridden in the inner classes, as no inheritance is involved.
    Like any other class member, they have class scope.
- can have any accessibility
- multiple objects of the inner classes can be associated with an object of an enclosing class at runtime
  for each instance of the outer class, there can exist many instances of a non-static inner class
- compiled to Outer$Inner.class , Outer$Inner$InnerOfInner.class
- can use the import statement to provide a shortcut for the names of non-static inner classes
- can extend other classes
- can themselves be extended
- case

```
class B{}
class A
{
    class C extends B{}
}
```

: If a name conflict arises,
the inherited member shadows the member with the same name in the enclosing class

- The compiler requires explicit reference
- can't say just the name in class C the compiler will complain: "Ambiguous name: inherited 'B.name' and outer scope 'A.name'—an explicit 'this' qualifier is required"
- this.name will refer to the one of superclass
- A.this.name will refer to the one of exclosing class

Ex1

```
class Outer
{
    public Inner makeInstance()
    {
        return new Inner(); //can use Inner() directly here
    }
    class Inner
    {
    }
}
public class Client
{
    public static void main(String args[])
    {
        Outer outRef = new Outer();
        Outer.Inner inRef = outRef.makeInstance();
        //an instance of a non-static inner class must be created in
        the context of an instance of the enclosing class
    }
}
```

Ex2

```
class Outer
{
    class Inner
    {
    }
}
public class Client
{
    public static void main(String args[])
    {
        Outer outRef = new Outer();
        Outer.Inner inRef = outRef.new Inner();
        Outer.Inner inRef2 = new Outer().new Inner();
    }
}
```

Ex3

```
class Outer
{
    static Inner i = new Outer().new Inner();
    Inner i2 = new Inner();
    class Inner
```

```
    {
    }
}
```

Local Classes

Anonymous Classes

- defined at the location they are instantiated
- do not have a name
- an instance of the class can only be created together with the definition

Ex

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
class FrameMouse extends Frame
{
        public static void main (String[] args)
        {
                FrameMouse fm = new FrameMouse();
        }
        public FrameMouse()
        {
                addWindowListener
                (
                        new WindowAdapter()
                        {
                                public void windowClosing(WindowEvent
                                e){dispose(); System.exit(0);}
                        }
                );
                setBackground(Color.pink);
                setSize(500,500);
                setVisible(true);
        }
}
```

Static Initializers

⇒ a block of code preceded by the word keyword ***static***.

- used to initialize a static variable in a class when simple initialization is not appropriate
- is not like a class method;
- does not have a formal parameter
- does not have a name
- does not return a value
- is not called from another part of the program
- When a class is loaded, the static initializer is automatically executed by the system before the ***main*** method is executed.
- A class may have a series of static initializers, one following the next in a sequence.
    - The system will execute all the static initializers in sequence before executing the main method.

Ex

```
public class Class1
{
    static {String s = "assfd";}
    public static void main(String args[]){}
}
```

Array

⇒ a data structure which defines an ordered collection of a fixed number of homogeneous data elements

- are objects
- all elements in the array have the **same data type**
- size is fixed
- is indexed from 0 (zero)
- arrayName.length → return the length of an array
- The declaration of the size of an array, may be performed at run-time, the size of the array is then fixed for for the duration of the executing program.
- If the index goes out of bounds (not within the limits of the array) then an *ArrayIndexOutOfBoundsException* is thrown.
- instance variable: length
- implicitly augment the inheritance hierarchy
- all array types implicitly extend the Object class
- Arrays of class types also extend the array type Object[]
- runtime: ArrayIndexOutOfBoundsException

One-dimensional arrays

Declaring:

Type[] arrayName

Type arrayName[]

- Note
    - int a[], b; → only “a” is an array
    - int[] a,b → both “a” and “b” are arrays
- only declared, not constructed or initialized
    - instance/static array → initialized to the default reference value null
    - no default initialization for local reference variables or local arrays

Constructing:

arrayName = new Type[noOfElement];

- min elements = 0
    - a common natural occurance of an array of length zero ⇒ array given as an argument to the main() method when a Java program is run without any program arguments
- all elements are initialized to the default value
- **anonymous arrays**

  new Type[]{}

- int[] name; → null
- int[] numbers = new int[5]; → storage space is allocated to an array using the keyword new
- int[] numbers = {1, 2, 3, 4};

  numbers[0] → 1
- Object[] array = {new String(), new String(), null};
- **int size = 5;**

  int[] numbers = new int[size];

```
//→ need to have the " = new int[size]" part
for (int index=0; index != size; index++)
{
        numbers[index] = new
        Integer(keyboard.readLine()).intValue();
}
```

- ```
  int[] numbers = {1, 2, 3, 4};
  for (int index=0; index != number.length; index++)
  {
        screen.println("cell " + index + "\t" + numbers[index]);
  }
  ```

Multidimensional arrays

- arrays in a multidimentional array need not have the same lengththe dimensions must be created from left to right
  - compile error: new int [][4]
- int[][] a ≡ int[] a[]
- int[][] A = {{1}, {1, 2}, {1, 2, 3}};
  - A.length → 3
  - A[1].length → 2
- ```
  int[][][] A =
  {
      { {0,1}, {1,2}, {2,3} },
      { {1,2}, {2,3}, {3,4} },
      { {2,3}, {3,4}, {4,5} },
      { {3,4}, {4,5}, {5,6} }
  };
  ```
- The declaration of an array may omit the lower-order dimensions, provided those dimensions that are declared are continuously described
  - new int[20][15][][]
  - // new int[20][][3] not OK
- ```
  for(int i=0; i != A.length; i++)
  {
      for(int j=0; j!= A[i].length; j++)
      {
          for(int z=0; z != A[i][j].length; z++)
          {
              screen.print(A[i][j][z] + "\t");
          }
      }
  }
  ```

java.lang.**String**

⇒ group of characters, that are stored as consecutive characters in the memory of a computer, with each character being represented by a 16-bit Unicode

String()
String(char[])
public String(char value[], int offset, int count)
- length = number of 16-bit Unicode characters
- offset argument : index of the first character of the subarray
- count : the length of the subarray

String(String)
String(StringBuffer)

- not a primitive data type
- A string literal in Java is delimited by double quotes.
- "ABC" or "\u0041\u0042\u0043"
- default = *null*
- shortcut method for initializing a string
  String objectName (= "argument-list";)
- If you wish to assign one string to another, then the assignment does not provide a copy of the value but merely a *reference* to the value
- + → a string concatenation operator
- String objects are immutable → the contents of the string cannot be changed
- public int **length()**
  - length = number of 16-bit Unicode characters
- public char **charAt**(int index)
- public void getChars(int srcBegin, int srcEnd, char dst[], int dstBegin)
  - The first character to be copied is at index srcBegin; the last character to be copied is at index srcEnd-1 (thus the total number of characters to be copied is srcEnd-srcBegin).
  - The characters are copied into the subarray of dst starting at index dstBegin and ending at index: dstbegin + (srcEnd-srcBegin) – 1.
- public boolean **equals**(Object anObject)
- public boolean **equalsIgnoreCase**(String anotherString)
- public int **compareTo**(String anotherString)
  - Returns:
    - the value 0 if the argument string is equal to this string;
    - a value less than 0 if this string is lexicographically less than the string argument
    - a value greater than 0 if this string is lexicographically greater than the string argument.
- public String substring(int beginIndex)
- public String substring(int beginIndex, int endIndex)
- public String concat(String str)
- public static String copyValueOf(char data[])
- public static String copyValueOf(char data[], int offset, int count)
- public char[] **toCharArray**()
- public String **toLowerCase**()
- public String **toUpperCase**()
- public static String **valueOf**(boolean/char/char[]/double/float d)
- public static String valueOf(char data[], int offset, int count)
- A call to the instance methods *toLowerCase()* and *toUpperCase()* on a string object will return a new string object, and will not modify the original string
  - If you intend to pass a string object as an argument, with the intention of changing the values of the string object → use the class *StringBuffer* to instantiate an
  ```

object that represents a string of characters
    - The characters in a *StringBuffer* object can be changed allowing the object to grow or shrink in length as necessary.
- Ex `StringBuffer data = new StringBuffer("abcde");`

Exception Handling

- Exception need not be caught in the same context that it was thrown in.
- The runtime behavior of the program determines which exceptions are thrown and how they are caught



- java.lang.Error
  - LankageError, ThreadDeath, VirtualMachineError
  - invariably never explicitly caught
  - usually irrecoverable
- java.lang.RuntimeException
  - usually due to program bugs that should not occur in the first place
  - more appropriate to treat them as faults in the program design, rather than merely catching them during program execution
- unchecked exceptions ⇒ RumtimeException, Error and their subclasses
  - a method is not obliged to deal with them
- checked exceptions ⇒ all exceptions except unchecked exceptions
    - Compiler ensures that
      if a method can throw a checked exception, directly or indirectly,
      then the method must explicitly deal with it
        - catch and take appropriate action
        - pass on to its caller
- If not explicitly caught and handled by the program, percolates upwards in the method activation stack, and is dealt with by the default exception handler ⇒ usually print s the name of the exception, with an explanatory message, folloed by the stack trace
- new exceptions are created by extending the Exception class or its subclasses → checked

try, catch, finally

```
try{}
catch(Exception1 e1){}
…
catch(Exceptionn en){}
finally{}
```

- the catch block and finally block must always appear in conjunction with a try block, and in the above order
- try
    - for each try block,
      there can be ≥ 0 catch blocks,
      but only one finally block
    - must be followed by either at least one catch block or one finally block
    - termination occurs when
        - encounter an exception → control is transferred to the catch block, if any.
            - if no catch block matches the thrown exception, control is transferred to the finally block
        - successful execution of the code inside → catch blocks are skipped and the control is transferred to the finally block, if one is specified
- catch
    - take exactly one argument
    - exception must be of the Throwable class or one of its subclasses
    - handle exception that is assignable to the reference type of the parameter
    - first matching catch is executed → all other catch blocks are skipped → control is transferred to finally block (regardless of whether the catch block itself throws an exception)
    - compile error: catch block for a superclass exception shadows the catch block for a subclass exception
      as the catch block of the subclass exception will never be executed
    - when exit catch block,
        - if there is any pending thrown exception that not handled → method is aborted → finally block executed, if any → exception propagated
        - if exception has been dealt with → finally block executed → normal execution resumes
- finally
    - if any code in the try block is executed, then the finally block is always executed
    - on exit, if there is any pending exception, the method is aborted and the exception propagated
- An exception which is thrown in a finally block overrules any previously unhandled exception, and is propagated in the usual way.

throw

throw ExceptionTypeObjectRef
throw new ExceptionType()

- when an exception is thrown,
  - normal execution is suspended
  - runtime system proceeds to find a catch block that can handle the exception
  - the search starts in the context of the current try block, propagating to any enclosing try blocks and through the method invocation stack to find a handler for the exception
  - Any associated finally block of a try block encountered along the search path is executed
  - If no handler is found, then the exception is dealt with by the default exception handler at the top level
  - If a handler is found, execution resumes with the code in its catch block

throws

- in method header
- explicitly propagate the exception to its caller
- a method can only throw those checked exceptions that are specified in its throws clause
- Overriding method: method definition in the subclass can only specify **all or subset** of the exception classes (including their subclasses) specified in the throws clause of the overridden method in the superclass