RELIANCE Tech Services
Anil Dhirubhai Ambani Group

Core Java

RELIANCE Tech Services
Anil Dhirubhai Ambani Group

Core Java

Introduction

Applets vs. applications

- Java applets
  - a program embedded into a web page
  - download and run on user's browser (or applet viewer)
  - internet programming
- Java applications
  - stand-alone programs – Java is a fully-fledged programming language
  - many Java class libraries for
    - GUI, graphics, networking
    - data structures
    - database connectivity
- we will be writing applications in this module.

First, a bit of background…

- 1st Generation – machine language
  - (raw machine code – lots of binary 0100101010001000111)
- 2nd Generation – assembly language
  - (mnemonic representation – short series of chars to represent binary)
- 3rd Generation – structured programming
  - (e.g. Pascal, C, C++, Java)
- 4th Generation – application specific
  - (SQL, Mathematica, RPG II, PostScript)
- 5th Generation – combining artificial intelligence
  - (best not mentioned…)

- The aim is to have a programming language that is as close as possible to natural speech – a bit like in ‘Star Trek’.

The Software Crisis

- The “software crisis”, recognized c.1969. - threatened the progress of the computer industry.
- People-time was and still is relatively expensive, machine-time is now very, very cheap.
- Programming was and still is very time intensive.
- Products need support - this is probably more than ever
- Software is complex; imagine creating a car with no drawings, specifications or planning.
- Amazingly, this is how a lot of the software in the past was created (some of it still is)!
- This situation had to end…

Towards Structured Programming

- As computer programs become larger and more complex, more errors are introduced.
- It has been estimated that there are 15 bugs in every 1000 lines of commercial code.
- Windows 2000 had 40 million lines of code!
- Most bugs are caused by poor memory management.
- Clearly there is a need for a structured programming language that helps in reducing the number of errors and speeds development for programming teams.
- C → C++ → Java
- Functional Programming → Object Orientation

“C makes it easy to shoot yourself in the foot; C++ makes it harder, but when it happens you tend to take off the whole leg!”

What is Object-Oriented code?

- Procedural coding is just a list of instructions.
- Object-oriented code has a lot of similarity with code from other procedural languages.
- Basically, a lot of the ‘words’ are the same but the way the words are put together (the structure) is different.
- Object-oriented coding does also contain lists of instructions but these lists are bound to specific objects.
- What an object actually represents is up to the programmer.

Simple Example

- A car is an object (in the real world) and in our program we can make a software object to represent it.
- Fundamentally, an object contains data and methods which can act on that data.
- The data we might want to have in our ‘software’ car could be things like: body colour, engine size, current speed, whether it has electric windows etc. Basically it’s up to you.
- The methods we might want our ‘car’ to have could be things like: accelerate, brake, respray body, open passenger window etc. Again, it’s up to you.
- Whatever you need your software car to model from the real world object must be present in your code.
- The OO syntax enables you to do this intuitively.

What technology does it replace?

- Procedural programming (C, Visual Basic, Fortran etc.)
- In procedural programming, functions were the most important part of the software.
- Where and how the data was stored was secondary (at best).
- Procedural code is process-oriented, OO code is data-oriented.
- In procedural programming, if your code is in error then it is relatively easy to fix but incorrect data may be *impossible* to fix!

Java (why use it?)

Advantages over C & C++

- WORA - Write Once, Run Anywhere (portable).
- Security (can run “untrusted” code safely).
- Robust memory management (opaque references, automatic garbage collection)
- Network-centric programming.
- Multi-threaded (multiple simultaneous tasks).
- Dynamic & extensible.
  - Classes stored in separate files
  - Loaded only when needed
  - Can dynamically extend itself to expand its functionality (even over a network and the Internet!)

How does it help?

- Large projects can be broken down into modules more easily.
- Aids understanding.
- Groupwork is easier.
- Less chance of data corruption.
- Aids reusability/extensibility.
- Maintaining code is far easier.
- Hides implementation details (just need to know what methods to call but no need to understand how the methods work to use them).

A simple Java application

```
import corejava.Console;
public class Hello  {
   public static void main(String args[])
   {
         String name; // this declares the variable “name”
         name = Console.readString ("What is your name?  ");
         System.out.println("Hello, " + name);
    }
}    /* end of program */
```

The Hello class and its method

- name of class is same as name of file (which has .java extension)
- body of class surrounded by { }
- this class has one method called main
  - all Java applications must have a main method in one of the classes
  - execution starts here
  - body of method within { }
- all other statements end with semicolon ;

Java keywords

- keywords appear in **bold**
  - reserved by Java for predefined purpose
  - don’t use them for your own variable, attribute or method names!
- **public**
  - visibility   could be **private**
- **static**
  - the main method belongs to the Hello class, and not an instance (object) of the class
- **void**
  - method does not return a value

Comments

- important for documentation!!!!
- ignored by compiler

```
//     single line (or part of line)

/*  multiple line comments go here
       everything between the marks
       is ignored */
```

- useful to ‘comment out’ suspect code or make notes

```
/**
 * These are used by the javadoc utility to create HTML
 * documentation files automatically.
 */
```

Variables and data types

- name is a variable of type String
- we have to declare variables before we use them
- unlike C, variables can be declared anywhere within block
- use meaningful names   numberOfBricks
- start with lower case
- capitalise first letter of subsequent words

Data types

- **int** 4 byte integer (whole number)
  - range -2147483648 to +2147483648
- **float** 4 byte floating point number
  - decimal points, numbers outside range of **int**
- **double** 8 byte floating point number
  - 15 decimal digits (float has 7) so bigger precision and range
- **char** 2 byte letter
- **String** string of letters
- **boolean** true or false (not 1 or 0)

Console input

- data input is difficult in Java
- methods for input in the library class corejava.Console
- we have to **import** corejava.Console to use it
- each book has its own methods for input!
  - Malik and Nair "Java programming" book uses raw Java which is difficult to read

Console methods

- general form:

  myVariable = Console.readType("Put a prompt here");

  - Console.readString("prompt")
  - Console.readInt("prompt")
  - Console.readDouble("prompt")
  - Console.readWord("prompt")
- gives error message if wrong type is input.
- Handy!

System output

- Java provides print methods in the class System.out (don’t need to import)
- println(name);
  - prints out what is stored in *name*, then goes to a new line
- print(name);
  - prints out what is stored in *name*, but does not start a new line
- print("My name is " + name);
  - put text in quotes
  - use + to print more than one item

Methods in Java

- methods break down large problems into smaller ones
- your program may call the same method many times
  - saves writing and maintaining same code
- methods take parameters
  - information needed to do their job
- methods can return a value
  - must specify type of value returned

Example method

- signature

```
public static int addNums(int num1, int num2)
{
        int answer = num1 + num2;
        return answer;
}
```

- body

Method signature

visibility [static] returnType methodName(parameterList)

- visibility:
  - **public**
    - accessible to other objects and classes
  - **protected**
    - accessible to classes which inherit from this one
  - **private**
- **static** keyword:
  - use when method belongs to class as whole
    - not object of the class

Method signature

visibility [static] returnType methodName(parameterList)

- return type:
  - specifies type of information returned
  - can be a simple type
    - **int**, **float**, **double**, **char**, String, **boolean**
  - or a class
  - if nothing returned, use keyword **void**
- method name:
  - use meaningful name which describes what method does!

Method signature

visibility [static] returnType methodName(parameterList)

- parameter list:
  - information needed by method
  - pairs of *type name*
  - examples:

    ```
    addNums(int num1, int num2)
    drawPerson(boolean isBald, String name, int numEarrings)
    ```

  - use empty brackets if method has no parameters

    ```
    printHeadings()
    ```

Method body

- signature

```
public static int addNums(int num1, int num2)
{
                int answer = num1 + num2;
                return answer;
}
```

- body

Method body

- use curly brackets to enclose method body
- all your code goes in here
  - write it so the method does what you intended
- last line should return a value of appropriate type
  - must match type in method header
  - nothing is executed after **return** statement
  - if method returns **void,** can omit **return** statement
    - method will automatically return at closing **}**

Calling a method

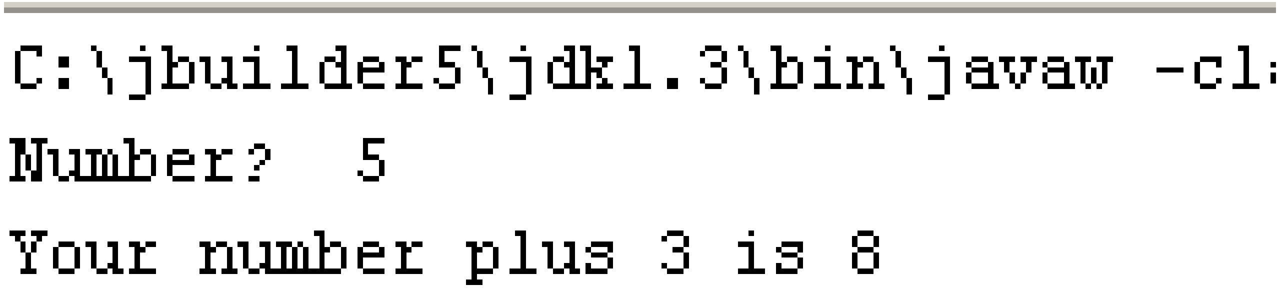
- methods will not run unless called from elsewhere
  - a statement in main() method could call another method
  - this method could call a third method .....
- class methods are called with the form:

  ClassName.methodName(parameters);

  - omit ClassName if called in same class
- method name and parameters must match the method signature
- if the method returns a value, it can be stored in a variable or passed to another method

Calling methods

```
public static void main(String args[])
{
    int input;
    input = Console.readInt("Number? ");
    System.out.print("Your number plus 3 is ");
    System.out.println(addNums(input, 3));
}
```



```
C:\jbuilder5\jdk1.3\bin\javaw -cl
Number?  5
Your number plus 3 is 8
```

Calling methods

- the previous example uses four methods
- from class Console:

  public static int readInt(String prompt)

  - store returned integer in variable input

- from System.out:

  public void print(String s)

  public void println(**int** x)

  - no returned value (void)

Calling methods

- from class JavaTest:

  public static int addNums(**int** num1, **int** num2)

  - ➢ pass returned integer to println() method
  - ➢ gets printed to screen
  - ➢ could also store it in another variable

  int answer = addNums(input, 3);

Features added in Java 1.2

- The following features are added in Java 1.2:
  - Java Swing
  - CORBA: Common Object Request Broker Architecture
  - Digital Certificates: ensures security policies
  - Collection API: for example: linked list, dynamic array

Features added in Java 1.3

- The following features are added in Java 1.3:
  - XML Processing
  - JDBC 3.0 API
  - Swing Drag and drop
  - Internationalization
  - Performance Improvement in Reflection APIs
  - JNDI
  - Java Print service API

Features added in Java 1.4

- The following features are added in Java 1.4:
  - New Security certificates
  - New Swing Features
  - Regular expressions
  - New I/O API
  - Logging
  - Secure Sockets
  - Assertions

Features added in Java 1.5

- The following features are added in Java 1.5:
  - Generics
  - Autoboxing / Unboxing
  - Enhanced for loop (“foreach”)
  - Type-safe enumerations
  - Variable number of arguments
  - Static import
  - Metadata (Annotations)

Getting Started

- Java software development involves the following steps:

1. Install Java - http://java.sun.com/j2se/1.5.0/
2. Write Myfirst.java program.

```
class Myfirst {
     public static void main(String args[]) {
     System.out.println("Welcome to Java ");          } }
```

1. Set PATH environment variable to the java bin folder.
   - For example: set PATH=C:\Program Files\Java\jdk1.5.0_07\bin
2. Compile the program.
   - C:\javac Myfirst.java
3. Execute the program.
   - C:\java Myfirst

Variable Argument List

- New feature added in J2SE5.0.
- Allows methods to receive unspecified number of arguments.
- An argument type followed by ellipsis(…) indicates variable number of arguments of a particular type.
  - *Variable-length* argument can take from *zero* to *n* arguments.
  - Ellipsis can be used only once in the parameter list.
  - Ellipsis must be placed at the end of the parameter list.

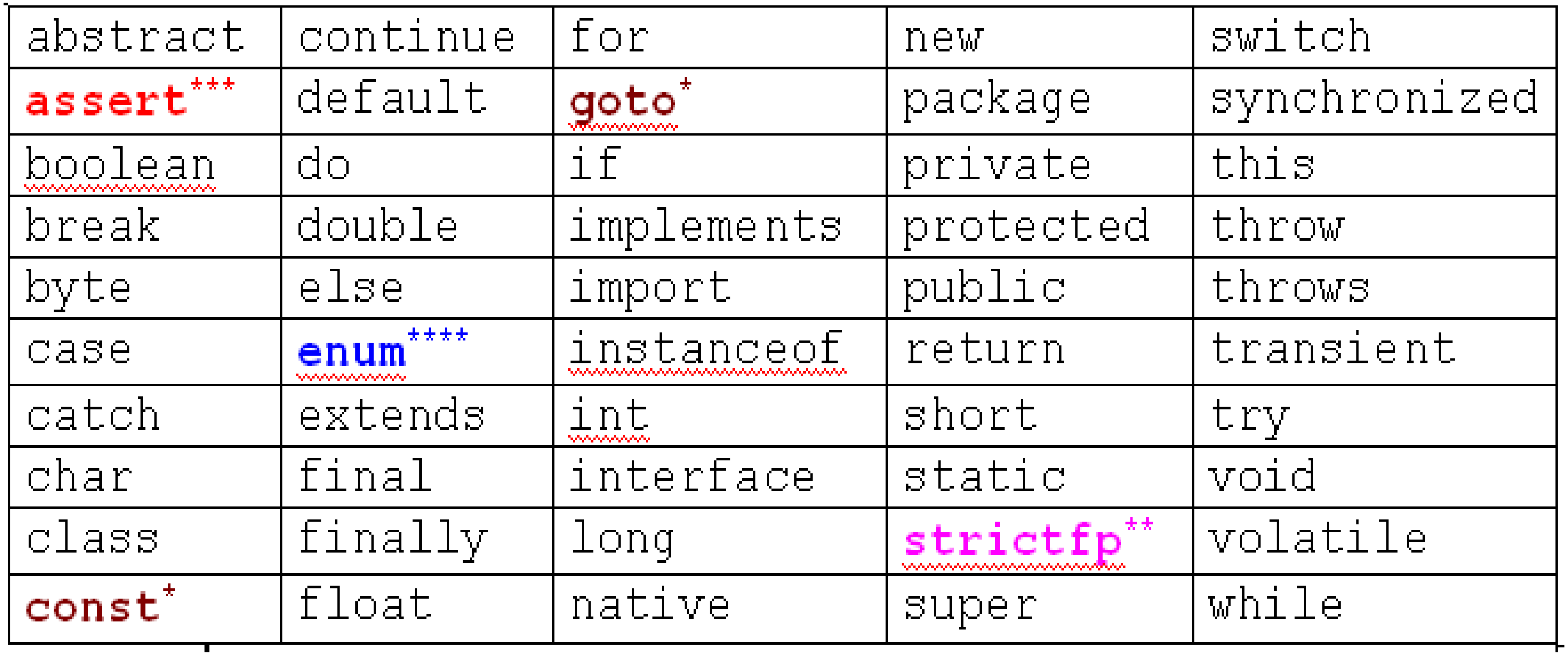
Variable Argument List (contd..)

```
void print(int a,int y,String...s)
  {
           //code
  }
```

- print(1,1,"XYZ") or print(2,5) or print(5,6,"A","B") invokes the above print function.
- //Invalid Code

```
void print(int a, int b…,float c)
{
  //code
}
```

Keywords in Java



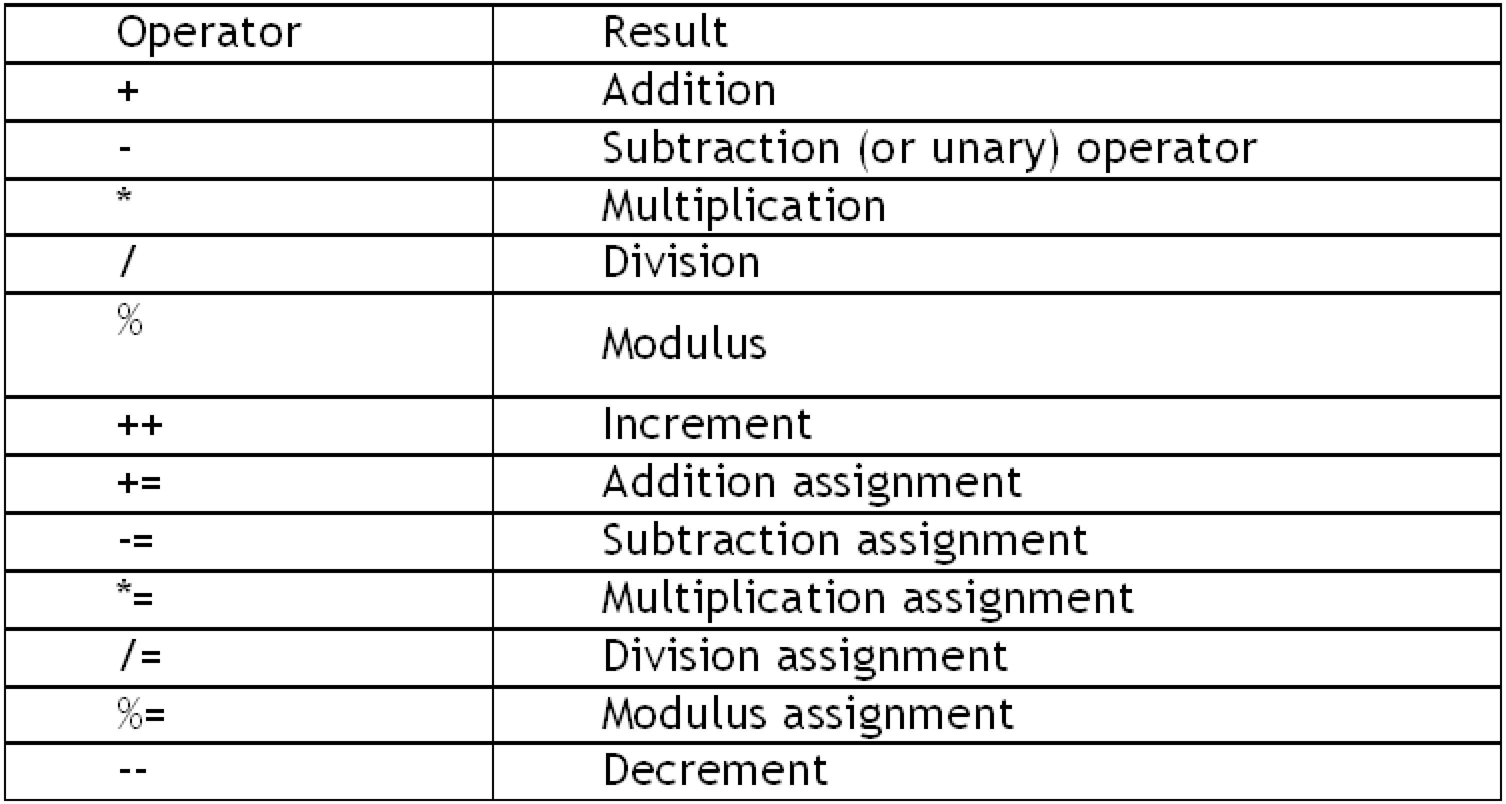
abstract	continue	for	new	switch
assert***	default	goto*	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum****	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp**	volatile
const*	float	native	super	while

- * not used *** added in 1.4
- ** added in 1.2 **** added in 5.0

Operators and Assignments Java

- Operators can be divided into following groups:
  - Arithmetic
  - Bitwise
  - Relational
  - Logical
  - *instanceOf* Operator

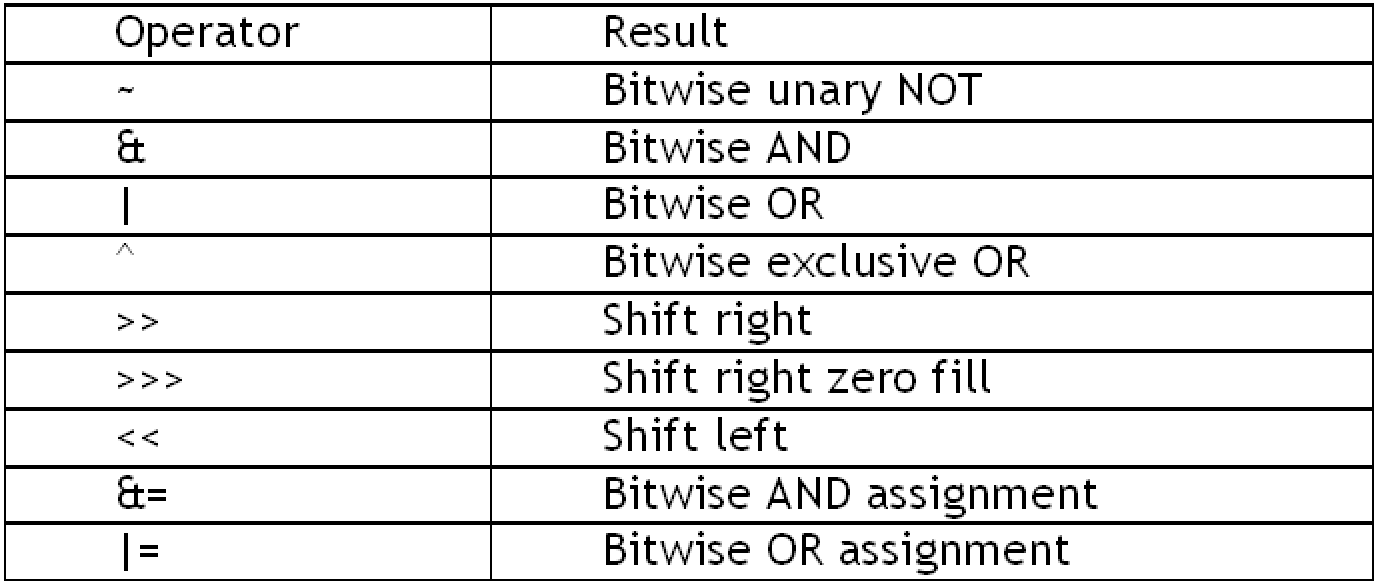
Arithmetic Operators



Operator	Result
+	Addition
-	Subtraction (or unary) operator
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
--	Decrement

Bitwise Operators

- Apply upon *int*, *long*, *short*, *char* and *byte* data types:



Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment

Relational Operators

- Determine the relationship that one operand has to another.
  - Ordering and equality.

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Logical Operators

Operator	Result
&&	Logical AND
||	Logical OR
^	Logical XOR
!	Logical NOT
==	Equal to
?:	Ternary if-then-else

instanceOf Operators

- The *instanceof* operator compares an object to a specified type
- Checks whether an object is:
  - An instance of a class.
  - An instance of a subclass.
  - An instance of a class that implements a particular interface.
  - The following returns *true*:

```
new String("Hello") instanceof String;
```

Control Statements

- Use control flow statements to:
  - Conditionally execute statements.
  - Repeatedly execute a block of statements.
  - Change the normal, sequential flow of control.
- Categorized into two types:
  - Selection Statements
  - Iteration Statements

Selection Statements

- Allows programs to choose between alternate actions on execution.
- “If” used for conditional branch:

```
if (condition) statement1;

else statement2;
```

- “Switch” used as an alternative to multiple “if’s”:

```
switch(expression){
case value1:    //statement sequence
     break;
case value2:    //statement sequence
     break; …
default:       //default statement sequence
}
```

Iteration Statements

- Allow a block of statements to execute repeatedly.
- *While* Loop:
  - ➢ Enters the loop if the condition is *true*.

```
while (condition)
{  //body of loop }
```

- *Do - While* Loop:
  - ➢ Loop executes at least once even if the condition is *false*.

```
do
{ //body of the loop } while (condition)
```

- *For* Loop:

```
for( initialization ; condition ; iteration)
{ //body of the loop }
```

Enhanced for Loop (foreach)

- New feature introduced in Java 5.
- Iterate through a collection or array.
  - Syntax:

```
for (variable : collection)
{ //code}
```

  - Example

```
int sum(int[] a)
{
 int result = 0;
for (int i : a)
result += i;
return result; }
```

Objects and Classes

- Classes:
  - A template for multiple objects with similar features.
  - A blueprint or the definition of objects.

```
class < class_name>
{
          type var1; …
          Type method_name(arguments )
          {
              body
          } …
} //class ends
```

- Objects:
  - Instance of a class.
  - Concrete representation of class.

Example

```
class Box{
   double width;
   double height;
   double depth;
   double volume(){
         return width*height*depth;
   } //method volume ends.
}//class box ends.
```

Example

```
class impl{
   public static void main(String a[])
   {
   //declare a reference to object
   Box b;
   //allocate a memory for box object.
   b = new Box();
   // call a method on that object.
   b.volume();  }
}
```

Types of Class Members

➢

- Default access members (No access specifier)
- Private members
- Public members
- Protected members

Memory Management

- Dynamic and Automatic
- No *Delete* operator
- Java Virtual Machine (JVM) de-allocates memory allocated to unreferenced objects during the garbage collection process.

Enhancement in Garbage Collector

- Garbage Collector:
  - Lowest Priority Daemon Thread
  - Runs in the background when JVM starts.
  - Collects all the unreferenced objects.
  - Frees the space occupied by these objects.
  - Call *System.gc()* method to “hint” the JVM to invoke the garbage collector.
    - There is no guarantee that it would be invoked. It is implementation dependent.

Memory Management

- All Java classes have *constructors*.
  - Constructors initialize a new object of that type.
- Default no-argument constructor is provided if program has no constructors.
  - Constructors:
    - Same name as the class.
    - No return type; not even void.

Finalize() Method

- Memory is automatically de-allocated in Java.
- Invoke *finalize()* to perform some housekeeping tasks before an object is garbage collected.
- Invoked just before the garbage collector runs:
  - protected void finalize()

Method Overloading

- Two or more methods within the same class share the *same* name.
- Their parameter declarations are different.
- Java implements compile-time polymorphism by:
    - Overloading Constructors
    - Overloading Normal Methods

Method Overloading

- Two or more methods within the same class share the *same* name.
- Their parameter declarations are different.
- Java implements compile-time polymorphism by:
  - ➢ Overloading Constructors
  - ➢ Overloading Normal Methods

Arrays

- A group of like-typed variables referred by a common name.
- Array declaration:
  - ➢ int arr [];

    arr = new int[10]
  - ➢ int arr[] = {2,3,4,5};
  - ➢ int two_d[][] = new int[4][5];
  - ➢ strWords = { "quiet", "success", "joy", "sorrow", "java" };

Creating Array Objects

- Arrays of objects too can be created:
  - Example 1:
    - Box Barr[] = new Box[3];
    - Barr[0] = new Box();
    - Barr[1] = new Box();
    - Barr[2] = new Box();
  - Example 2:
    - String[] Words = new String[2];
    - Words[0]=new String(“Bombay”);
    - Words[1]=new String(“Pune”);

Declaring Type Safe Enums

- Problem: (pre-J2SE 5.0)

  public static final int SEASON_WINTER = 0;

  Public static final int SEASON_SUMMER = 1;

  Not type safe (any integer will pass)

  No namespace (SEASON_*)

  Brittleness (how do add value in-between?)

  Printed values uninformative (prints just int values)

- Solution: New type of class declaration

  enum type has public, self-typed members for each enum constant

Declaring Type Safe Enums (contd..)

- Permits variable to have only a few pre-defined values from a given list.
- Helps reduce bugs in the code.
  - Example:

```
enum CoffeeSize { BIG, HUGE, OVERWHELMING };
CoffeeSize cs = CoffeeSize.BIG;
```

- *cs* can have values *BIG*, *HUGE* and *OVERWHELMING* only.

Declare Enums Outside a Class

- *enum* can be declared with only a *public* or *default* modifier.

```
enum CoffeeSize { BIG, HUGE, OVERWHELMING } // this cannot be
// private or protected
class Coffee {
CoffeeSize size;
}
public class CoffeeTest1 {
public static void main(String[] args) {
Coffee drink = new Coffee();
drink.size = CoffeeSize.BIG; // enum outside class
} }
```

Declare Enums Inside a Class

```
class Coffee2 {
enum CoffeeSize {BIG, HUGE, OVERWHELMING }
CoffeeSize size;
}
public class CoffeeTest2 {
public static void main(String[] args) {
Coffee2 drink = new Coffee2();
drink.size = Coffee2.CoffeeSize.BIG; // enclosing class
// name required
} }
```

- *enums* are not strings or integer type. In the above example BIG is of type *CoffeeSize*.

Enums

- *enum* cannot be declared in functions.
- A semicolon after an enum is optional.

```
public class CoffeeTest1 {
public static void main(String[] args) {
enum CoffeeSize { BIG, HUGE, OVERWHELMING } // WRONG!
  Cannot
// declare enums
// in methods
Coffee drink = new Coffee();
drink.size = CoffeeSize.BIG;
}}
```

Declare Constructors, Methods and Variables in an enum

- Add constructors, instance variables, methods, and a *constant specific class body*.
  - ➢ Example:

```
enum CoffeeSize {
BIG(8), HUGE(10), OVERWHELMING(16);
// the arguments after the enum value are "passed"
// as values to the constructor
CoffeeSize(int ounces) {
this.ounces = ounces; // assign the value to
// an instance variable }
```

Example

```
private int ounces; // an instance variable each enum
public int getOunces() {
return ounces; } }
class Coffee {
CoffeeSize size; // each instance of Coffee has-a
public static void main(String[] args) {
Coffee drink1 = new Coffee();
drink1.size = CoffeeSize.BIG;
Coffee drink2 = new Coffee();
drink2.size = CoffeeSize.OVERWHELMING;
System.out.println(drink1.size.getOunces()); // prints 8
System.out.println(drink2.size.getOunces()); // prints 16  } }
```

OOPS Features in Java

- Inheritance
- Polymorphism

Interfaces (cont’d)

- An interface in Java is like an abstract class, but it does not have any fields or constructors, and all its methods are abstract.

```
public interface Dance{
    DanceStep getStep (int i);
    int getTempo ();
    int getBeat (int i);
}
```

- “public abstract” is not written because all the methods are public abstract.

Interfaces (cont’d)

- We must “officially” state that a class **implements** an interface.
- A concrete class that implements an interface must supply all the methods of that interface.

```
public class Waltz implements Dance{
    ...
    // Methods:
    public DanceStep getStep (int i)  { ... }
    public int getTempo ()  { return 750; }
    public int getBeat (int i)  { ... }
    ...
}
```

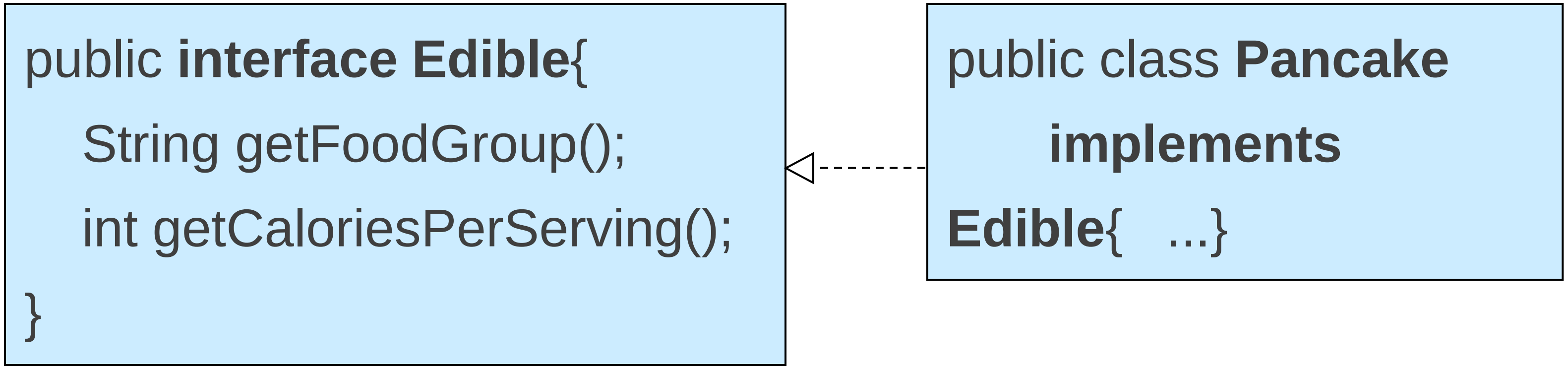
Interfaces (cont’d)

- A class can implement several interfaces.
- Like an abstract class, an interface supplies a secondary data type to objects of a class that implements that interface.
- You can declare variables and parameters of an interface type.

```
➢    Dance d = new Waltz( );
```

- Polymorphism fully applies to objects disguised as interface types.

Interfaces (cont’d)



```
public interface Edible{
    String getFoodGroup();
    int getCaloriesPerServing();
}
```

```
public class Pancake
      implements
Edible{   ...}
```

```
public class Breakfast
{
    private int myTotalCalories = 0;
    ...
    public void eat (Edible obj, int servings)
    {
      myTotalCalories +=
           obj.getCaloriesPerServing () * servings
    }
    ...
}
```

- Polymorphism: the correct method is called for any specific type of **Edible**, e.g., a **Pancake**

➢Similarities

Classes	Interfaces
• A superclass provides a secondary data type to objects of its subclasses.	• An interface provides a secondary data type to objects of classes that implement that interface.
• An abstract class cannot be instantiated.	• An interface cannot be instantiated.

➢Similarities

Classes	Interfaces
• A concrete subclass of an abstract class must define all the inherited abstract methods.	• A concrete class that implements an interface must define all the methods specified by the interface.
• A class can extend another class. A subclass can add methods and override some of its superclass’s methods.	• An interface can extend another interface (called its *superinterface*) by adding declarations of abstract methods.

➢Differences

Classes	Interfaces
• A class can extend only one class.	• A class can implement any number of interfaces.
• A class can have fields.	• An interface cannot have fields (except, possibly, some public static final constants).
• A class defines its own constructors (or gets a default constructor).	• An interface has no constructors.

➢Differences

Classes	Interfaces
• A concrete class has all its methods defined. An abstract class usually has one or more abstract methods.	• All methods declared in an interface are abstract.
• Every class is a part of a hierarchy of classes with Object at the top.	• An interface may belong to a small hierarchy of interfaces, but this is not as common.

Inheritance

- Major pillars of OO approach.
- Allows creation of hierarchical classification.
- Advantage is reusability of the code:
  - A class, once defined and debugged, can be used to create further derived classes.
- Extend existing code to adapt to different situations.

Inheritance (contd…)

- Use inherited members with the *super* keyword:
  - super(); // calls parent class constructor.
  - super.overriden();// calls a base class overridden method.

Inheritance (contd..)

```
Class Base {
 public void meth1() { System.out.println(“Base”);            } }
Class Derived extends Base {
    ✓public void meth1() {
    ✓      super.meth1();
    ✓      System.out.println(“Derived”); } }
class Test {
  public static void main(String args[]){
  Derived d1=new Derived();
  d1.meth1();
  } }
```

Polymorphism

- An objects ability to decide what method to apply to itself depending on where it is in the inheritance hierarchy.
- Can be applied to any method that is inherited from a *super class*.
- Design and implement easily extensible systems.

Method Overriding

- In a class hierarchy, when a method in a subclass has the same *name* and *type signature* as a method in its super class, then the subclass method overrides the super class method.
- Overridden methods allow Java to support run-time polymorphism.

Other Modifiers

- Abstract Method
- Abstract Class
- Final Variable
- Final Method
- Final Class

Abstract Method

- Methods do not have implementation.
- Methods declared in interfaces are always abstract.
  - Example:

```
abstract int  add(int a, int b);
```

Abstract Class

- Provides common behavior across a set of subclasses.
- Not designed to have instances that work.
- One or more methods are declared but may not be defined
- Advantages:
  - Code reusability.
  - Help at places where implementation is not available.

Abstract Class (cont…)

- Declare any class with even one method as abstract as *abstract*.
- Cannot be instantiated.
- Use *Abstract* modifier for:
  - Constructors
  - Static methods
- Abstract class' subclasses should implement all methods or declare themselves as *abstract*.
- Can have concrete methods also.

Final Variable

- Variable declared as *final* acts as a constant.
- Once initialized, it’s value cannot be changed.
- Example:
  - final int i = 10;

Final Method

- Method declared as final cannot be overridden in subclasses.
- Their values cannot change their value once initialized.
  - Example:

```
class A  {
public final int add (int a, int b)  {
return a+b;   }}
```

Final Class

- Cannot be sub-classed at all.
- Examples: *String* and *StringBuffer* classes in Java.
- A class or method cannot be abstract and final at the same time.

Static

- Static modifier can be used in conjunction with:
  - A variable
  - A method
  - A class

Static variable

- Static variable is shared by all the class members.
- Used independently of objects of that class.
- Static members can be accessed before an object of a class is created, by using the class name:
  - Example: static int intNum1 = 3;

Static Methods

- Restrictions:
  - Can only call other static methods.
  - Must only access other static data.
  - Cannot refer to *this* or *super* in any way.
  - Can access non-static variables and methods:
    - Explicit instance variables should be made available to the method.
    - Method *main()* is a static method. It is called by JVM.

Nested Classes

- Class within another class.
- Scope is bounded by the scope of the enclosing class.
- Use to reflect and enforce the relationship between two classes.

Types of Nested Classes

- Nested classes are of two types:
  - Static:
    - Access members of the enclosing class through an object.
    - Cannot refer to members of the enclosing class directly.
  - Non-static:
    - Also called as *Inner* classes.
    - Access to all members, including private members, of the class in which they are nested.
    - Can refer to the members of enclosing class in the same way as non-static members.

Anonymous Inner Class

- Do not have a name.
- Defined at the location they are instantiated using additional syntax with the *new* operator.
- Used to create objects “on the fly” in contexts such as:
  - Method return value.
  - Argument in a method call.
  - Variable initialization.

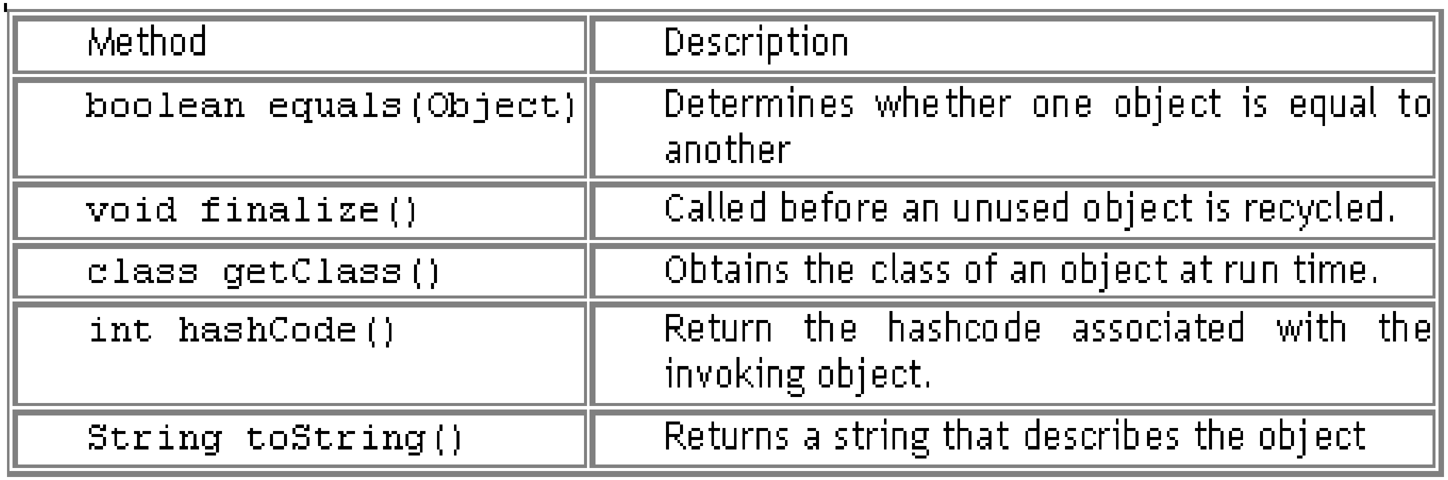
The Object Class

- Cosmic super class.
- Ultimate ancestor
  - Every class in Java implicitly extends Object.
- Object type variables can refer to objects of any type:
  - Example:

```
Object obj = new Emp();
```

- Object Class Methods:
  - void finalize()
  - Class getClass()
  - String toString()

Object Class Methods

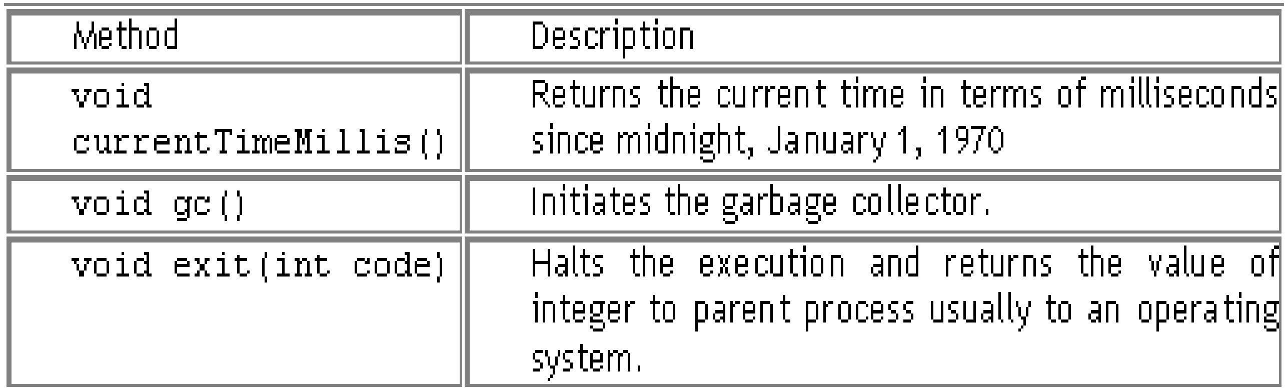


Method	Description
`boolean equals(Object)`	Determines whether one object is equal to another
`void finalize()`	Called before an unused object is recycled.
`class getClass()`	Obtains the class of an object at run time.
`int hashCode()`	Return the hashcode associated with the invoking object.
`String toString()`	Returns a string that describes the object

The System Class

- Used to interact with any of the system resources.
- Cannot be instantiated.
- Contains a methods and variables to handle system I/O.
- Facilities provided by the System class:
  - Standard input
  - Standard output
  - Error output streams

The System Class (contd..)



Method	Description
`void currentTimeMillis()`	Returns the current time in terms of milliseconds since midnight, January 1, 1970
`void gc()`	Initiates the garbage collector.
`void exit(int code)`	Halts the execution and returns the value of integer to parent process usually to an operating system.

String Handling

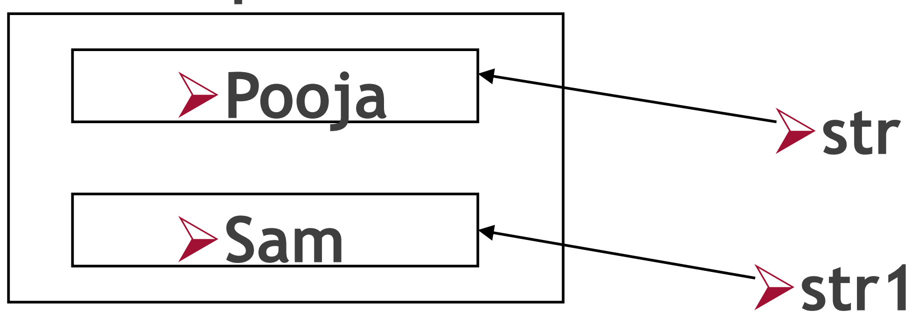
- String is handled as an object of class String and not as an array of characters.
- String class is a better and a convenient way to handle any operation.
- One main restriction is that once an object of this class is created, the contents cannot be changed.

String Handling – Important Methods

- length(): length of string.
- indexOf(): searches an occurrence of a char, or string within other string.
- substring(): Retrieves substring from the object.
- trim(): Removes spaces.
- valueOf(): Converts data to string.

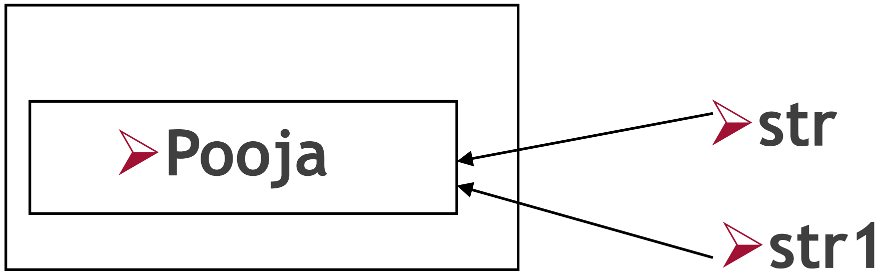
The String Class

- String str = new String(“Pooja”);
- String str1 = new String(“Sam”);
- Heap Stack



- Pooja
- Sam
- str
- str1

- String str = new String(“Pooja”);
- String str1 = str;



- Pooja
- str
- str1

String Concatenation

- Use a “+” sign to concatenate two strings:
  - String Subject = "Core " + "Java"; -> Core Java
  - *String concatenation* operator if one operand is a string:
    - String a = "String"; int b = 3; int c=7
    - System.out.println(a + b + c); -> String37
  - *Addition* operator if both operands are numbers:
    - System.out.println(a + (b + c)); -> String10

String Concatenation (contd..)

- public String concat(String s)
  - Used to concatenate a string to an existing string.
  - String x = "Core ";
  - System.out.println(x=x.concat(" Java"));
  - Output -> "Core Java”

String Comparison

```
class EqualsNotEqualTo {
  public static void main(String args[]) {
        String str1 = "Hello";
        String str2 = new String(str1);
        System.out.println(str1 + " equals " + str2 + " -> " +
                    str1.equals(str2));
        System.out.println(str1 + " == " + str2 + " -> " + (str1 ==str2));
  }
}
Output :  Hello equals Hello -> true
          Hello == Hello -> false
```

StringBuffer Class

- Use the following to make ample modifications to character strings:
  - *java.lang.StringBuffer*
  - java.lang.StringBuilder
- Many string object manipulations end up with a many abandoned string objects in the *String pool*
  - String Objects are immutable.

```
StringBuffer sb = new StringBuffer("abc");
sb.append("def");
System.out.println("sb = " + sb); // output is "sb = abcdef"
```

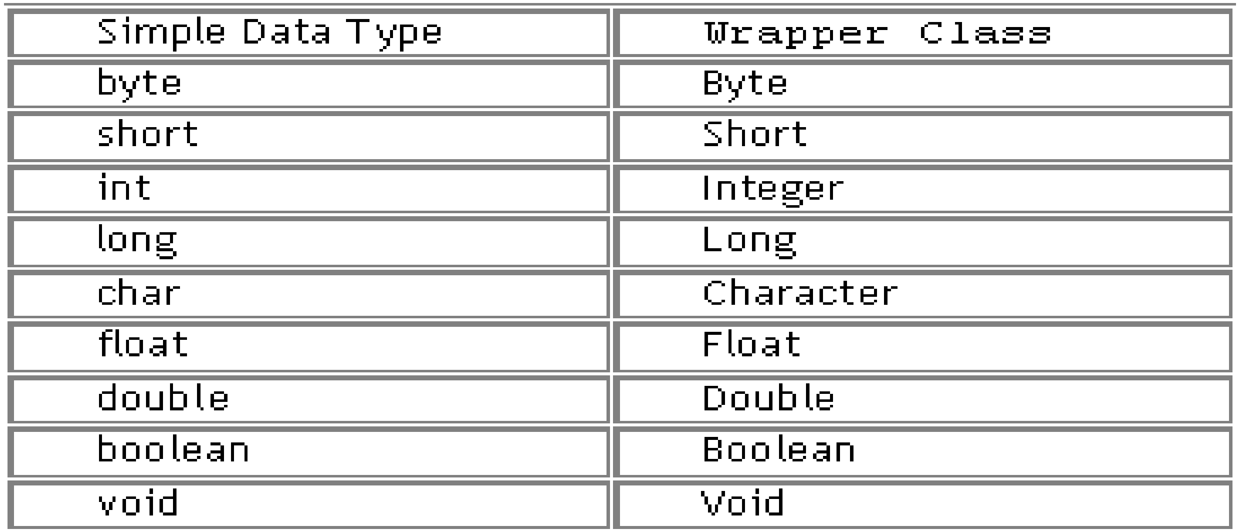
StringBuilder Class

- Added in Java 5.
- Exactly the same API as the *StringBuffer* class, except:
  - It is not thread safe.
  - It runs faster than StringBuffer.

```
StringBuilder sb = new StringBuilder("abc");
sb.append("def").reverse().insert(3, "---");
System.out.println( sb ); // output is "fed---cba"
```

Wrapper Classes

- Correspond to primitive data types in Java.
- Represent primitive values as objects.
- Wrapper objects are immutable.



Simple Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
char	Character
float	Float
double	Double
boolean	Boolean
void	Void

Casting for Conversion of Data type

- Casting operator converts one variable value to another where two variables correspond to two different data types.

```
variable1 = (variable1) variable2
```

- Here, *variable2* is typecast to *variable1*.
- Data type can either be a *reference* type or a *primitive* one.

Casting Between Primitive Types

- When one type of data is assigned to another type of variable, *automatic type conversion* takes place if:
  - Both types are compatible.
  - Destination type is larger than the source type.
  - No explicit casting is needed (widening conversion).

```
int a=5; float b; b=a;
```

- If there is a possibility of data loss, explicit cast is needed:

```
int i = (int) (5.6/2/7);
```

Casting Between Reference Types

- One class types involved must be the same class or a subclass of the other class type.
- Assignment to different class types is allowed only if a value of the class type is assigned to a variable of its superclass type.
- Assignment to a variable of the subclass type needs explicit casting:

```
String StrObj = Obj;
```

- Explicit casting is not needed for the following:

```
String StrObj = new String(“Hello”);
Object  Obj = StrObj;
```

Casting Between Reference Types (contd..)

- Two types of reference variable castings:
  - Downcasting:

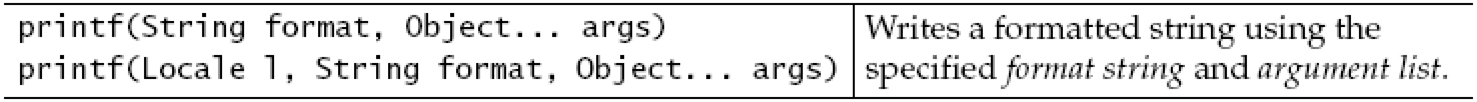
```
Object Obj = new Object ( );
String StrObj = (String) Obj;
```

  - Upcasting:

```
String StrObj = new String(“Hello”);
Object  Obj = StrObj;
```

Simple Formatted I/O and Scanner

- Classes *PrintStream* and *PrintWriter* provide following convenience methods for formatting output:



`printf(String format, Object... args)`
`printf(Locale l, String format, Object... args)`
	Writes a formatted string using the specified *format string* and *argument list*.

- *Format string syntax* provides support for layout justification and alignment, common formats for numeric, string, and date/time data, and locale-specific output.

Simple Formatted I/O

- *printf* has following formatting capabilities
  - Round floating-point values to given number of decimal places.
  - Align a column of numbers with decimal points that appear one above the other.
  - Right and left justification of outputs.
  - Insert literal characters at precise locations in a line of output.

Simple Formatted I/O (contd..)

- Represent floating-point numbers in exponential format.
- Represent integers in *octal* and *hexadecimal* format.
- Display all data types with fixed-size field widths and precisions.
- Display dates and times in various formats.

Scanner

- Scanner - A simple scanning API
  - Converts text into primitives or strings.

```
/* Reading from the console. */
import java.util.Scanner;
import static java.lang.System.out;
public class ConsoleInput {
public static void main(String[] args) {
// Create a Scanner which is chained to System.in, i.e. to the console.
```

Scanner (contd..)

```
Scanner lexer = new Scanner(System.in);
// Read a list of integers.
int[] intArray = new int[3];
out.println("Input a list of integers (max. " + intArray.length +
  "):");
for (int i = 0; i < intArray.length;i++)
intArray[i] = lexer.nextInt();
for (int i : intArray)
out.println(i);
// Read names
String firstName;
String lastName;
```

Scanner (contd..)

```
String name;
String repeat;
do {
lexer.nextLine(); // Empty any input still in the current line
System.out.print("Enter first name: ");
firstName = lexer.next();
lexer.nextLine();
System.out.print("Enter last name: ");
lastName = lexer.next();
```

Streams and Files

Objectives:

- Learn the basic facts about Java’s IO package
- Understand the difference between text and binary files
- Understand the concept of an input or output stream
- Learn about handling exceptions

Files

- A file is a collection of data in mass storage.
- A data file is not a part of a program’s source code.
- The same file can be read or modified by different programs.
- The program must be aware of the format of the data in the file.

Files (cont'd)

- The files are maintained by the operating system.
- The system provides commands and/or GUI utilities for viewing file directories and for copying, moving, renaming, and deleting files.
- The operating system also provides basic functions, callable from programs, for reading and writing directories and files.

Text Files

- A computer user distinguishes text (“ASCII”) files and “binary” files. This distinction is based on how you treat the file.
- A text file is assumed to contain lines of text (for example, in ASCII code).
- Each line terminates with a newline character (or a combination, carriage return plus line feed).

Text Files

- Examples:
  - Any plain-text file, typically named something.txt
  - Source code of programs in any language (for example, Something.java)
  - HTML documents
  - Data files for certain programs, (for example, fish.dat; any file is a data file for some program.)

Binary Files

- A “binary” file can contain any information, any combination of bytes.
- Only a programmer / designer knows how to interpret it.
- Different programs may interpret the same file differently (for example, one program displays an image, another extracts an encrypted message).

Binary Files

- Examples:
  - Compiled programs (for example, Something.class)
  - Image files (for example, something.gif)
  - Music files (for example, something.mp3)
- Any file can be treated as a binary file (even a text file, if we forget about the special meaning of CR-LF).

Streams

- A stream is an abstraction derived from sequential input or output devices.
- An input stream produces a stream of characters; an output stream receives a stream of characters, “one at a time.”
- Streams apply not just to files, but also to IO devices, Internet streams, and so on.

Streams (cont'd)

- A file can be treated as an input or output stream.
- In reality file streams are buffered for efficiency: it is not practical to read or write one character at a time from or to mass storage.
- It is common to treat text files as streams.

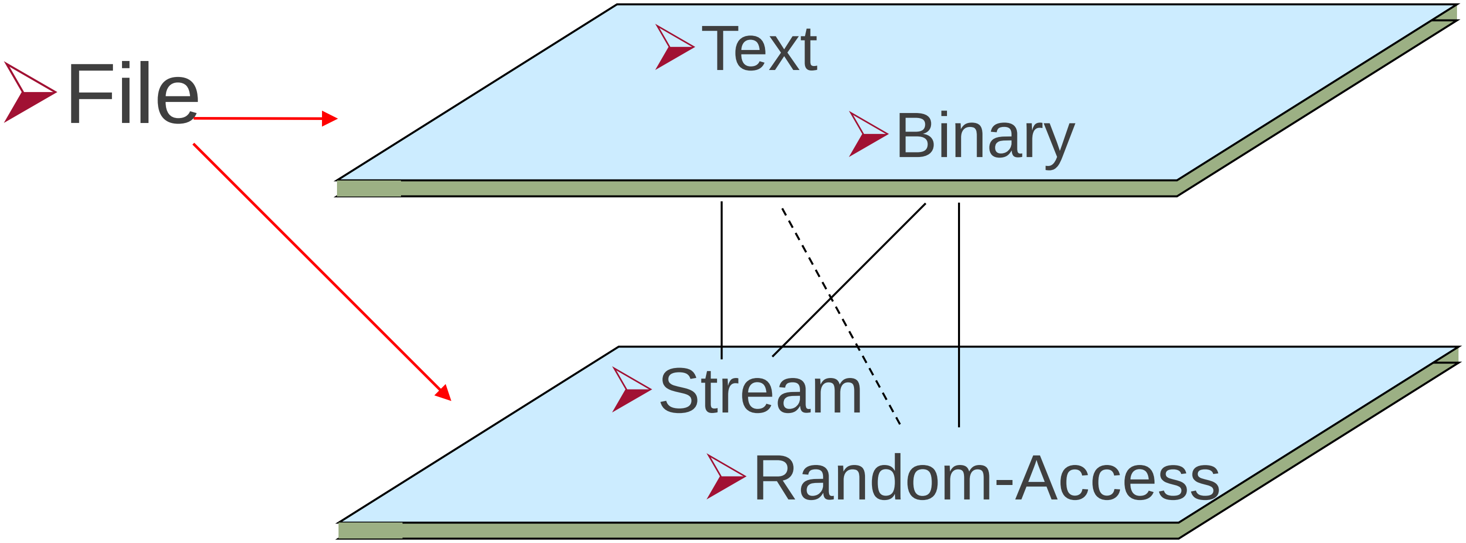
Random-Access Files

- A program can start reading or writing a random-access file at any place and read or write any number of bytes at a time.
- “Random-access file” is an abstraction: any file can be treated as a random-access file.
- You can open a random-access file both for reading and writing at the same time.

Random-Access Files (cont’d)

- A binary file containing fixed-length data records is suitable for random-access treatment.
- A random-access file may be accompanied by an “index” (either in the same or a different file), which tells the address of each record.
- Tape : CD == Stream : Random-access

File Types: Summary



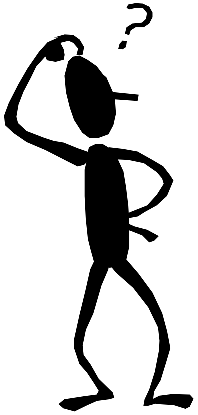
- ——— common use- - - - - -
- possible, but
- not as

java.io

BufferedInputStream
BufferedOutputStream
BufferedReader
BufferedWriter
ByteArrayInputStream
ByteArrayOutputStream
CharArrayReader
CharArrayWriter
DataInputStream
DataOutputStream
File
FileDescriptor
FileInputStream
FileOutputStream
FilePermission
FileReader
FileWriter
FilterInputStream
FilterOutputStream
FilterReader
FilterWriter

InputStream
InputStreamReader
LineNumberInputStream
LineNumberReader
ObjectInputStream
ObjectInputStream.GetField
ObjectOutputStream
ObjectOutputStream.PutField
ObjectStreamClass
ObjectStreamField
OutputStream
OutputStreamWriter
PipedInputStream
PipedOutputStream
PipedReader
PipedWriter
PrintStream
PrintWriter
PushbackInputStream
PushbackReader

RandomAccessFile
Reader
SequenceInputStream
SerializablePermission
StreamTokenizer
StringBufferInputStream
StringReader
StringWriter
Writer



- How do I read an **int** from a file?

java.io (cont’d)

- Uses four hierarchies of classes rooted at Reader, Writer, InputStream, OutputStream.
- InputStream/OutputStream hierarchies deal with bytes. Reader/Writer hierarchies deal with chars.
- Has a special stand-alone class RandomAccessFile.
- The Scanner class has been added to java.util in Java 5 to facilitate reading numbers and words.

java.io.File

- The **File** class represents a file (or folder) in the file directory system.

```
➢String pathname = "../Data/words.txt“;
➢File file = new File(pathname);
```

- Methods:

```
String getName()
String getAbsolutePath()
long length()
boolean isDirectory()
File[ ] listFiles()
```

Reading from a Text File

```
String pathname = "words.txt";
File file = new File(pathname);
Scanner input = null;
try
{
   input = new Scanner(file);
}
catch (FileNotFoundException ex)
{
   System.out.println("*** Cannot open " + pathname
                                                  + " ***");
   System.exit(1);  // quit the program
}
```

- Tries to open the file

Scanner Methods

```
boolean hasNextLine()
String nextLine()
boolean hasNext()
String next()
boolean hasNextInt()
int nextInt()
boolean hasNextDouble()
double nextDouble()
void close()
```

- Reads one word

Writing to a Text File

```
String pathname = "output.txt";
File file = new File(pathname);
PrintWriter output = null;
try  {
   output = new PrintWriter(file);
}catch (FileNotFoundException ex)
{
   System.out.println("Cannot create " + pathname);
   System.exit(1);  // quit the program
}
output.println(...);
output.printf(...);
output.close();
```

- Required to flush the output buffer

The Java Collections Framework

Overview

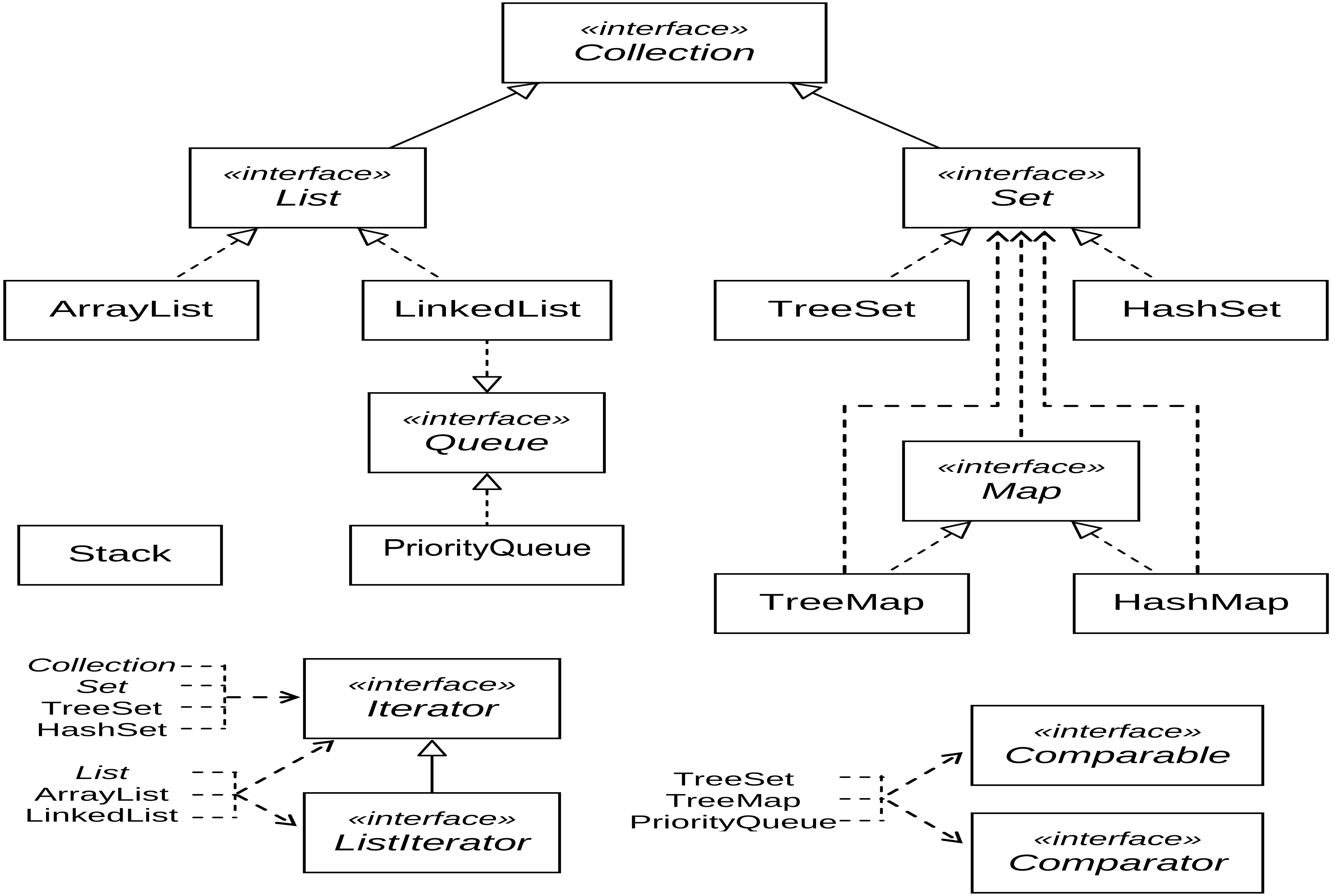
- *Framework* (in software): a general system of components and architectural solutions that provides development tools to programmers for use with a relatively wide range of applications.
- *Collection*: (hmm...) any collection of elements

Overview (cont'd)

- *Collection*, *Iterator*
- Lists, *ListIterator*
  - *List*
  - ArrayList
  - LinkedList
- Stack
- *Queue*, PriorityQueue
- Sets
  - *Set*
  - TreeSet
  - HashSet
- Maps
  - *Map*
  - TreeMap
  - HashMap

All these interfaces and classes are part of the **java.util** package. Names of interfaces are in italics.

Overview (cont’d)



Overview (cont'd)

- A collection holds references to objects (but we say informally that it "holds objects").
- A collection can contain references to two equal objects (a.equals (b)) as well as two references to the same object (a == b).
- An object can belong to several collections.
- An object can change while in a collection (unless it is immutable).

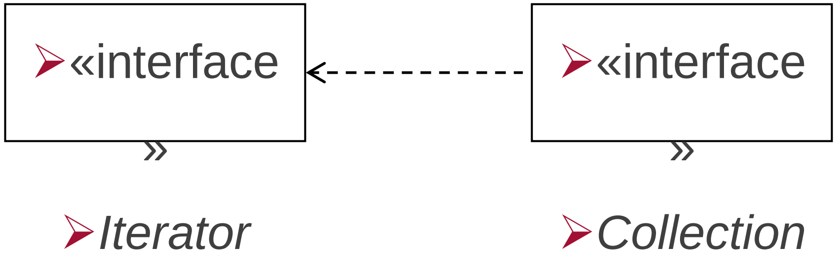
Overview (cont’d)

- Starting with Java 5, a collection holds objects of a specified type. A collection class’s or interface’s definition takes object type as a parameter:
  - *Collection<**E**>*
  - *List<**E**>*
  - Stack<***E***>
  - *Set<**E**>*

Because collections work with different types, these are called *generic collections* or *generics*

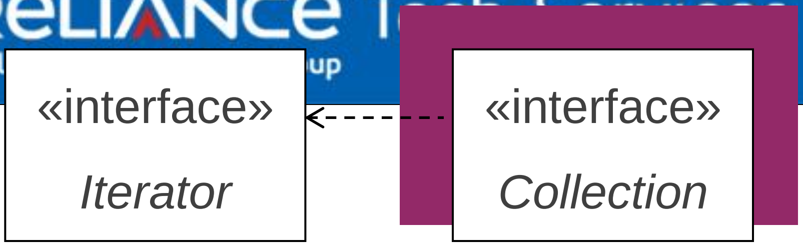
- A map takes two object type parameters:
  - Map<***K***, ***V***>

Collection, Iterator



- *Collection* interface represents any collection.
- An iterator is an object that helps to traverse the collection (process all its elements in sequence).
- A collection supplies its own iterator(s), (returned by collection's **iterator** method); the traversal sequence depends on the collection.

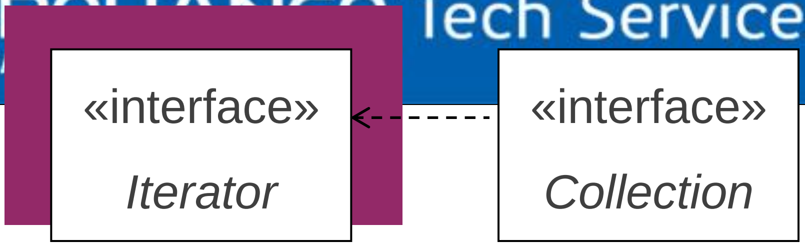
Collection<E>



```
boolean isEmpty ()

int size ()

boolean contains (Object obj)

boolean add (E obj)

boolean remove (E obj)

Iterator<E> iterator ()

//   ... other methods
```

- Supplies an iterator for this collection

Iterator<E>

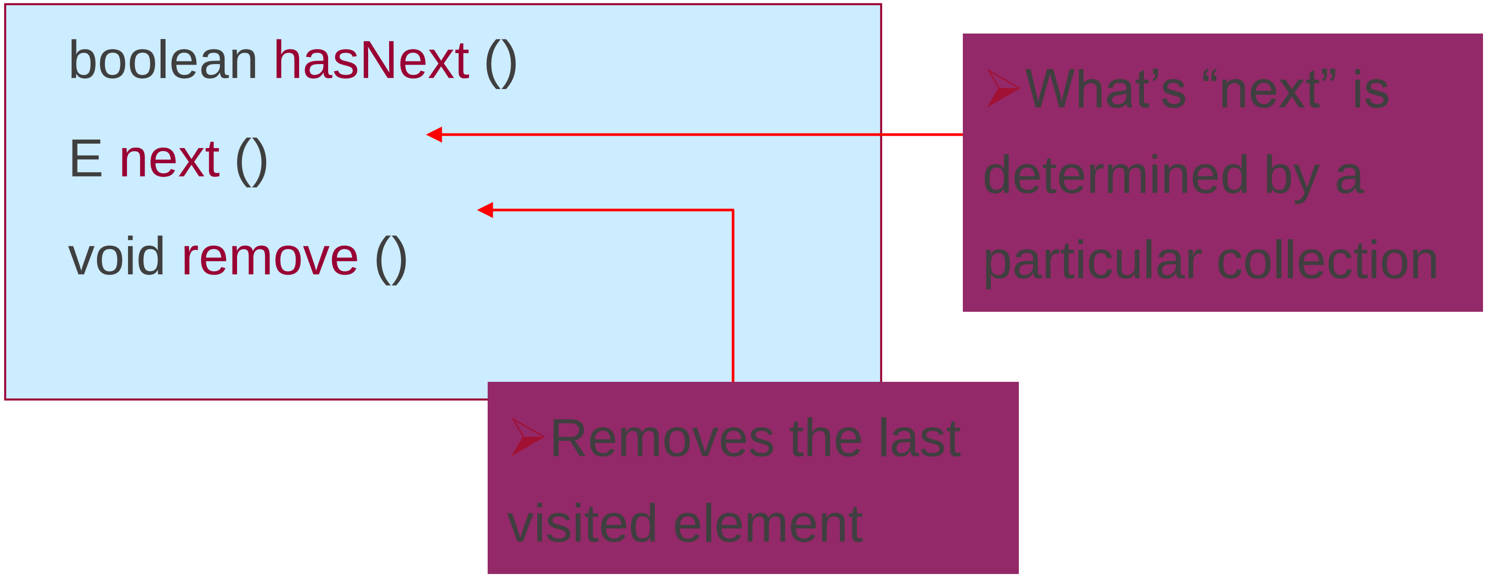


«interface»

Iterator

«interface»

Collection



boolean hasNext ()

E next ()

void remove ()

What's "next" is determined by a particular collection

Removes the last visited element

Iterator ⇔ "For Each" Loop

```
Collection<String> words = new ArrayList<String>();
...
```

```
for (String word : words)
{
    < ... process word >
}
```

```
Iterator<String> iter =
                words.iterator();
while (iter.hasNext ())
{
    String word = iter.next ();
    < ... process word >
```

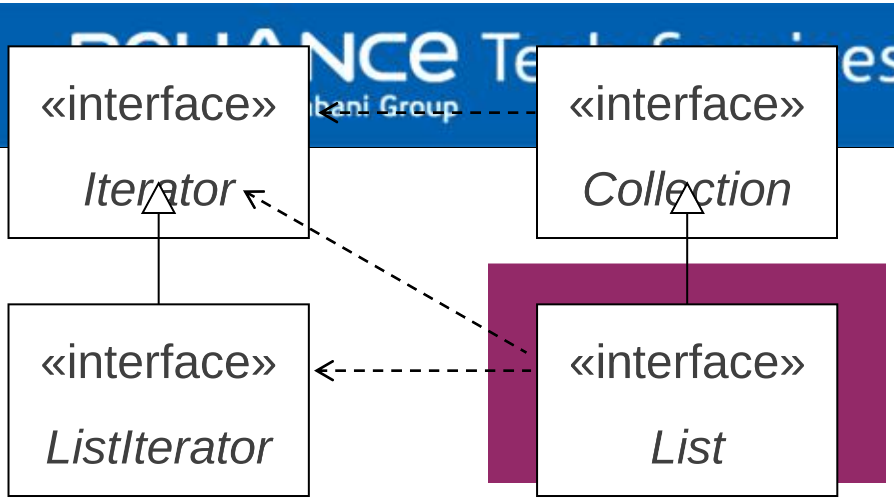
- A "for each" loop is a syntactic shortcut that replaces an iterator

Lists, *ListIterator*

- A list represents a collection in which all elements are numbered by indices:

  $a_0, a_1, \ldots, a_{n-1}$
- java.util:
  - *List* interface
  - ArrayList
  - LinkedList
- *ListIterator* is an extended iterator, specific for lists (*ListIterator* is a subinterface of *Iterator*)

List<E>

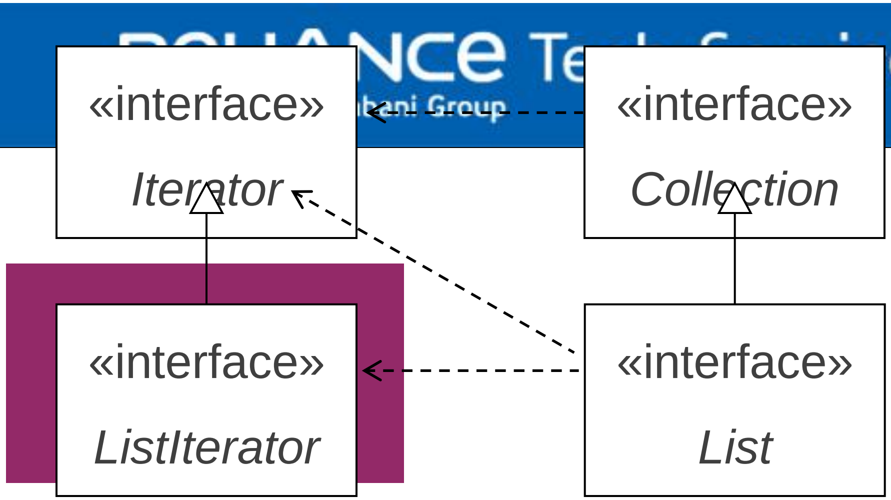


```
// All Collection<E> methods, plus:
E get (int i)
E set (int i, E obj)
void add (int i, E obj)
E remove (int i)
int indexOf (Object obj)
ListIterator<E> listIterator ()
ListIterator<E> listIterator (int i)
```

- These methods are familiar from **ArrayList**, which implements **List**

- Returns a **ListIterator** that starts iterations at index **i**

ListIterator<E>



«interface»
Iterator

«interface»
Collection

«interface»
ListIterator

«interface»
List

```
// The three Iterator<E> methods, plus:
int nextIndex ()
boolean hasPrevious ()
E previous ()
int previousIndex ()
void add (E obj)
void set (E obj)
```

- Can traverse the list backward
- Can add elements to the list (inserts after the last visited element)
- Can change elements (changes the last visited element)

ArrayList vs. LinkedList (cont'd)

	ArrayList	LinkedList
get(i) and **set(i, obj)**	***O*(1)**	*O*(*n*)
add(i, obj) and **remove(i)**	*O*(*n*)	*O*(*n*)
add(0, obj)	*O*(*n*)	***O*(1)**
add(obj)	***O*(1)**	***O*(1)**
contains(obj)	*O*(*n*)	*O*(*n*)

Stacks

- A stack provides temporary storage in the LIFO (Last-In-First-Out) manner.
- Stacks are useful for dealing with nested structures and branching processes:
  - pictures within pictures
  - folders within folders
  - methods calling other methods
- Controlled by two operations: *push* and *pop*.
- Implemented as `java.util.Stack<`*E*`>` class

Queues

- A queue provides temporary storage in the FIFO (First-In-First-Out) manner
- Useful for dealing with events that have to be processed in order of their arrival
- java.util:
  - *Queue* interface
  - LinkedList (implements Queue)

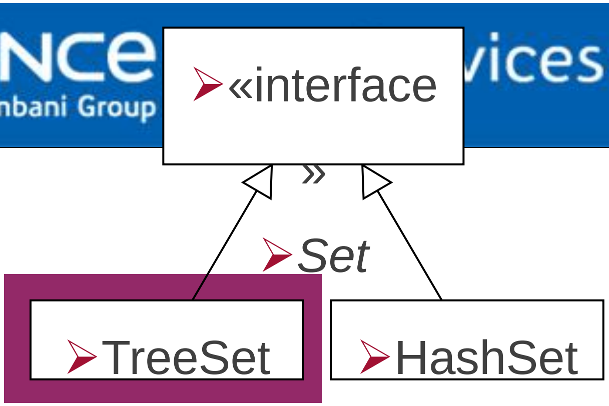
Priority Queues

- In a priority queue, items are processed NOT in order of arrival, but in order of priority.
- java.util:
  - *Queue* interface
  - PriorityQueue (implements *Queue*)

Sets

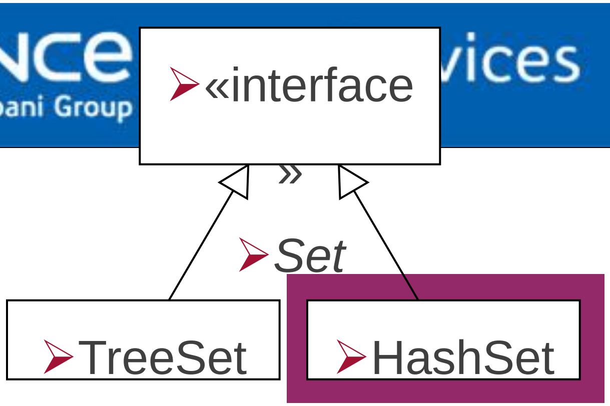
- A set is a collection without duplicate values
- What is a “duplicate” depends on the implementation
- Designed for finding a value quickly
- java.util:
  - *Set* interface
  - TreeSet
  - HashSet

TreeSet<*E*>



- Works with Comparable objects (or takes a comparator as a parameter)
- Implements a set as a *Binary Search Tree* (Chapter 23)
- contains, add, and remove methods run in *O*(log *n*) time
- Iterator returns elements in ascending order

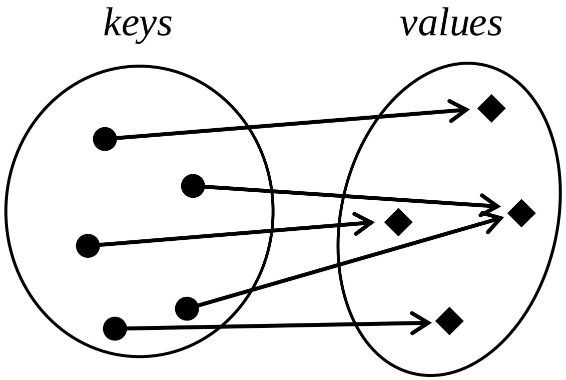
HashSet<*E*>



- Works with objects for which reasonable **hashCode** and **equals** methods are defined
- Implements a set as a *hash table* (Chapter 24)
- **contains**, **add**, and **remove** methods run in *O*(1) time
- Iterator returns elements in no particular order

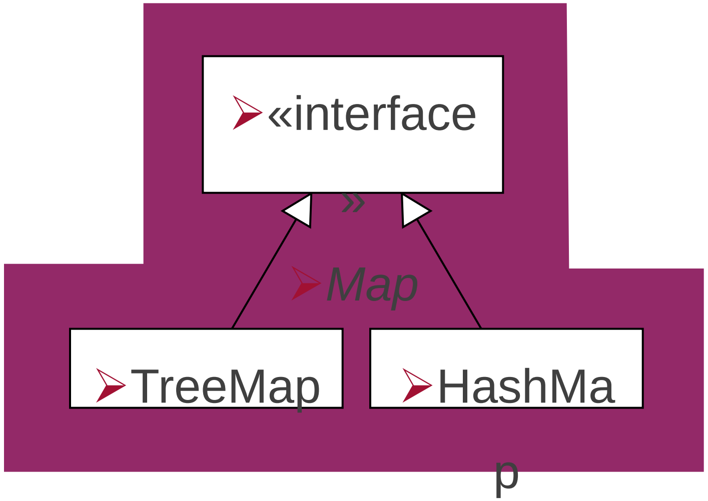
Maps

- A *map* is not a collection; it represents a correspondence between a set of keys and a set of values
- Only one value can correspond to a given key; several keys can be mapped onto the same value

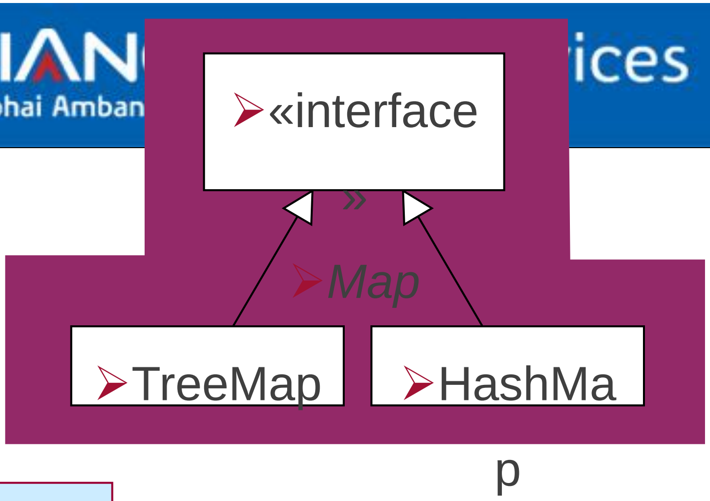


Maps (cont'd)

- java.util:
  - *Map* interface
  - TreeMap
  - HashMap



Map<K, V>



«interface

»

Map

TreeMap

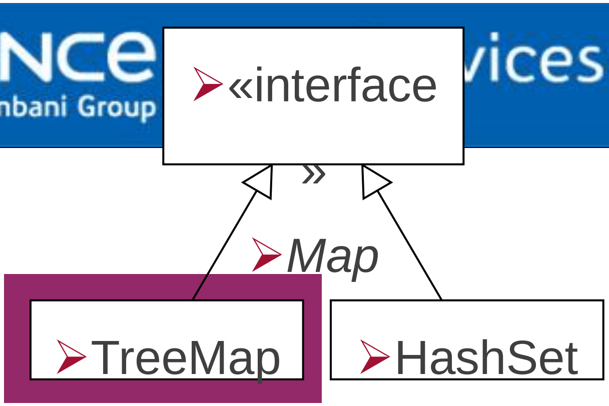
HashMa

p

```
boolean isEmpty ()

int size ()

V get (K key)

V put (K key, V value)

V remove (K key)

boolean containsKey (Object key)

Set<K> keySet ()
```

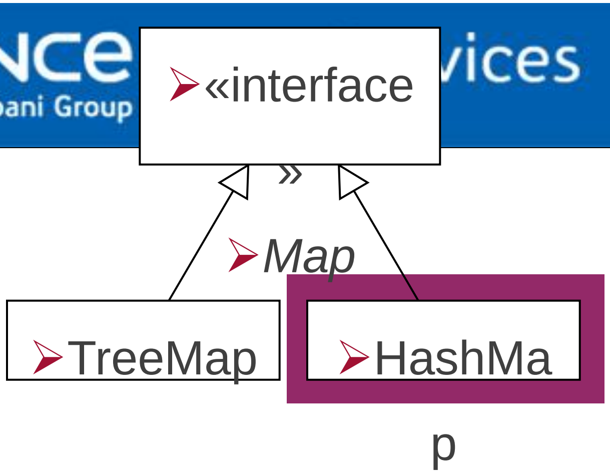
Returns the set of all keys

TreeMap<*K*,*V*>



- Works with **Comparable** keys (or takes a comparator as a parameter)
- Implements the key set as a *Binary Search Tree* (Chapter 23)
- **containsKey**, **get**, and **put** methods run in *O*(log *n*) time

HashMap<*K*,*V*>

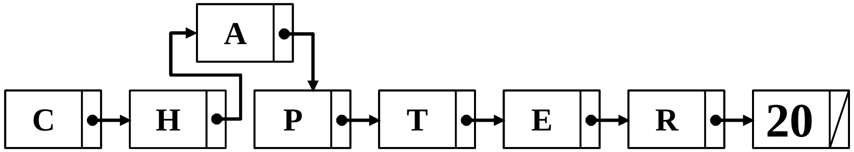


- Works with keys for which reasonable hashCode and equals methods are defined
- Implements the key set as a *hash table* (Chapter 24)
- containsKey, get, and put methods run in *O*(1) time

Example:

traversing all key-value pairs in a map

```
import java.util.*;
   ...
   Map<Integer, String> presidents =
             new TreeMap<Integer, String> ();

   presidents.put (1, “George Washington”);
   ...
   for (Integer key :  presidents.keySet() )  {
      String name = presidents.get (key);
      System.out.println (key + " : " + name);
   }
```



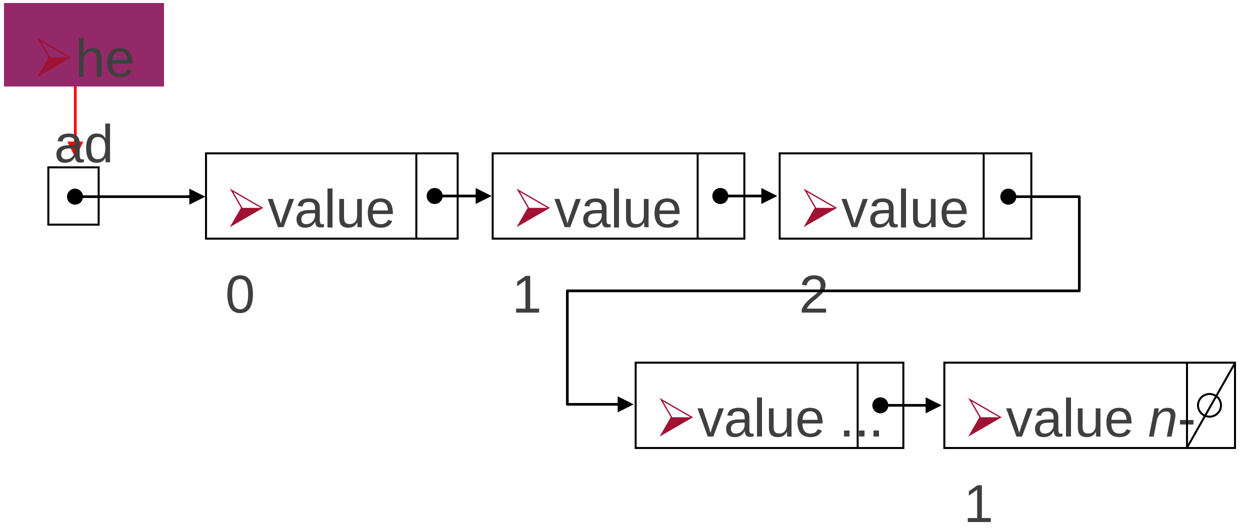
Lists and Iterators

Objectives:

- Learn to work with ListNode objects and do-it-yourself linked lists
- Understand singly-linked list, linked list with a tail, circular list, and doubly-linked list
- Learn to implement iterators

Singly-Linked List

- Each node holds a reference to the next node
- In the last node, **next is null**
- A linked list is defined by a reference to its first node (often named **head** or **front**)



Singly-Linked List (cont'd)

```
public class ListNode
{
   private Object value;
   private ListNode next;

   public ListNode (Object v, ListNode nx)
   { value = v; next = nx; }

   public Object getValue ( )   { return value; }
   public ListNode getNext ( )   { return next; }
   public void setValue (Object v)   { value = v; }
   public void setNext (ListNode nx)   { next = nx; }
```

- Represents a node of a singly-linked list

- A reference to the next node

Singly-Linked List – Example 1

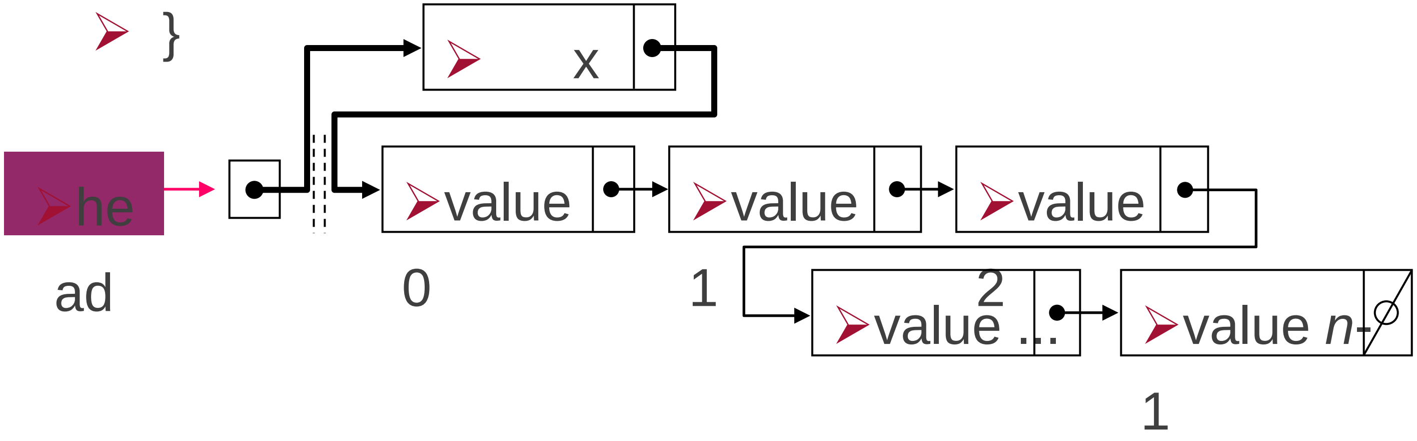
- Append x at the head of a linked list and return the head of the new list.

➢ public ListNode append (ListNode head, Object x)

➢ {

➢ return new ListNode (value, head);

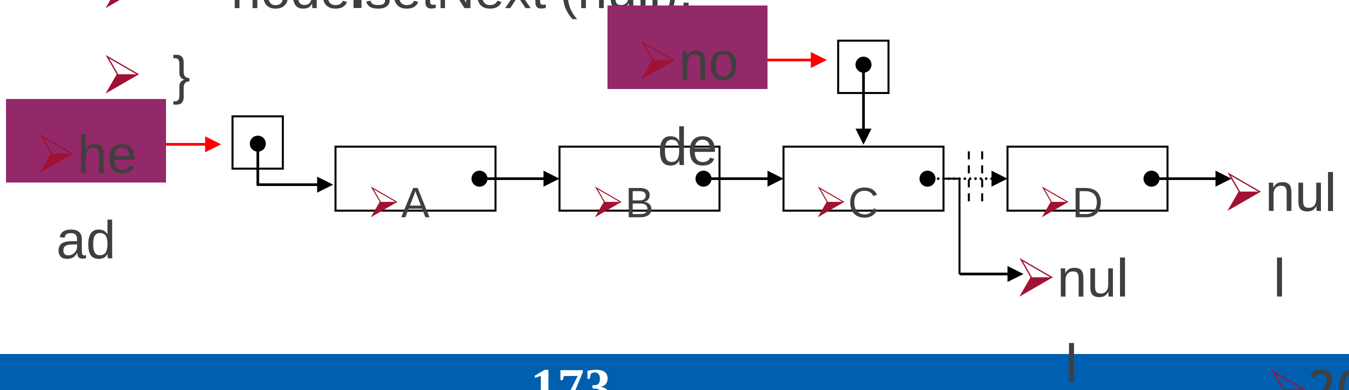
➢ }



Singly-Linked List – Example 2

- Assuming the list has at least two nodes, remove the last node.

```
public void removeLast (ListNode head)
{
    ListNode node = head;
    while (node.getNext ().getNext () != null)
        node = node.getNext ( );
    node.setNext (null);
}
```



Singly-Linked List – Traversal

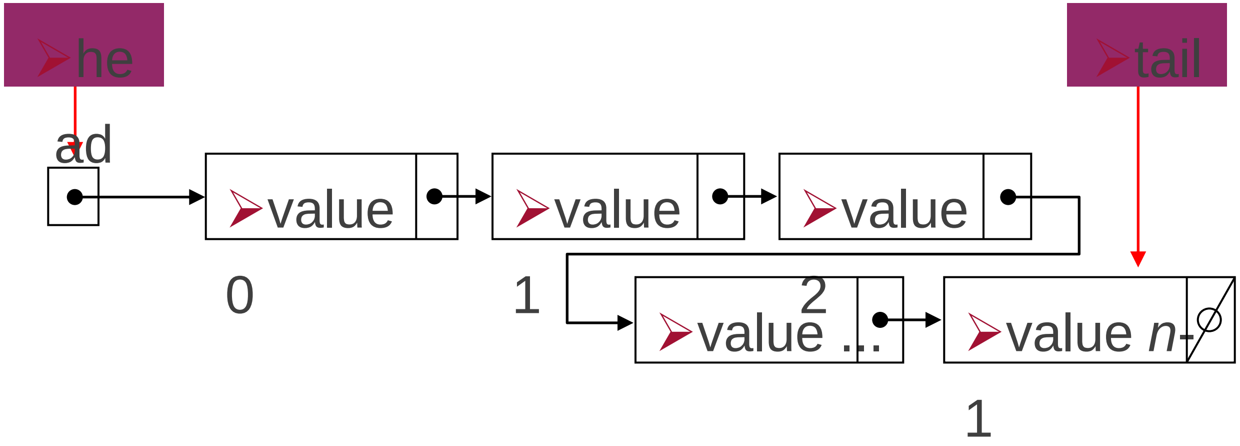
```
➢ public void printList (ListNode head)
➢ {
➢     for (ListNode node = head;  node != null;
➢                                             node = node.getNext
())
➢         System.out.println (node.getValue ());
➢ }
```

Do-it-Yourself Iterator (cont'd)

- public class **SinglyLinkedListIterator implements Iterator<Object>**
- {
- private ListNode nextNode;
- public SinglyLinkedListIterator (ListNode head) { **nextNode = head**; }
- public boolean hasNext ()
- { return nextNode != null; }
- public Object next ()
- {
- if (nextNode == null)
- throw new NoSuchElementException ();
- public void remove ()
- Object
- { throw new UnsupportedOperationException(); }
- **nextNode = nextNode.getNext ()**;

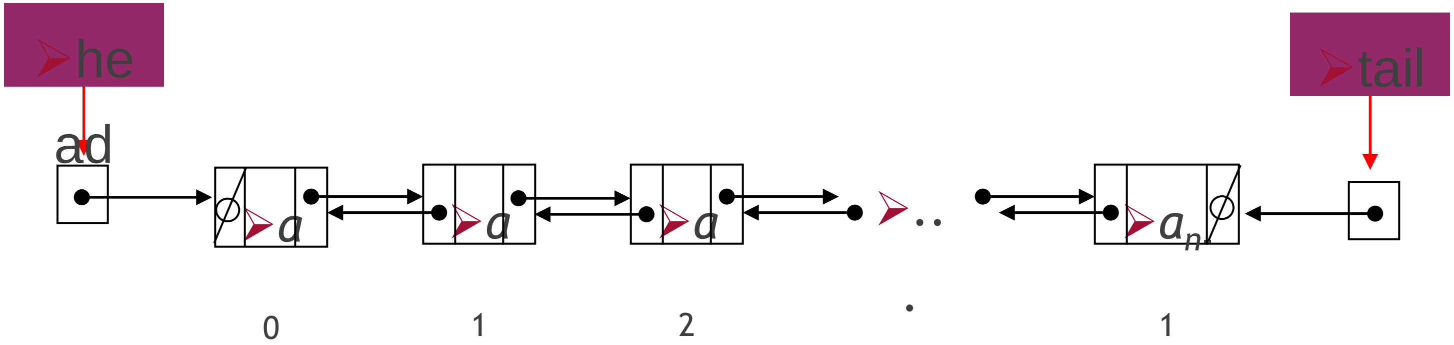
Singly-Linked List with a Tail

- Keeps a reference to the last node
- Suitable for implementing a queue



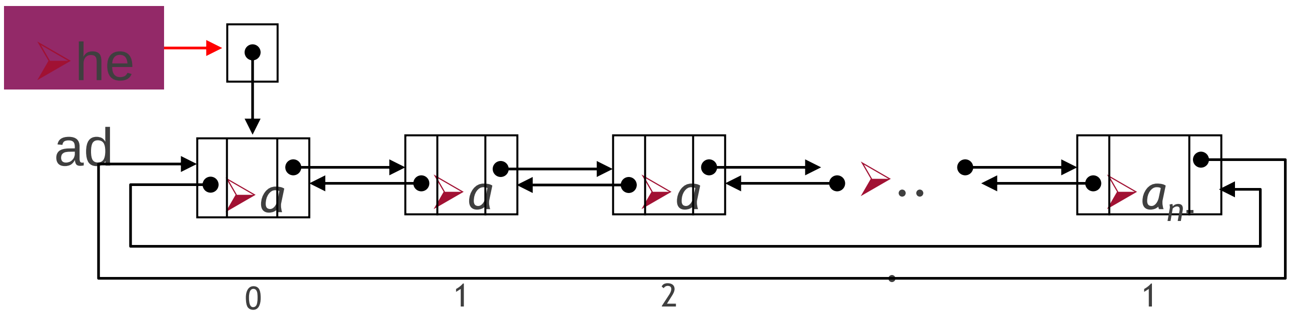
Doubly-Linked List

- Each node has references to the next and previous nodes
- In the last node, next is null; in the first node, previous is null.
- Can be traversed backward



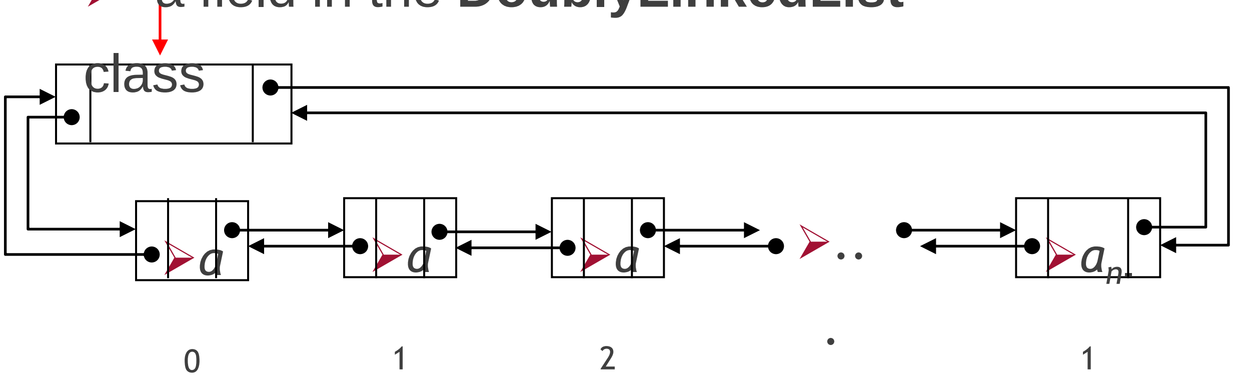
Doubly-Linked Circular List

- **next** in the last node points to the first node
- **previous** in the first node points to the last node



- That’s how java.util.LinkedList is implemented

- **private ListNode2 header;**
- a field in the **DoublyLinkedList** class



Reflections in Java

Java looking at Java

- One of the unusual capabilities of Java is that a program can examine itself
  - You can determine the class of an object
  - You can find out all about a class: its access modifiers, superclass, fields, constructors, and methods
  - You can find out what what is in an interface
  - Even if you don’t know the names of things when you write the program, you can:
    - Create an instance of a class
    - Get and set instance variables
    - Invoke a method on an object
    - Create and manipulate arrays

What is reflection for?

- In “normal” programs you don’t need reflection
- You *do* need reflection if you are working with programs that process programs
- Typical examples:
  - A class browser
  - A debugger
  - A GUI builder
  - An IDE, such as BlueJ, Netbeans oe eclipse
  - A program to grade student programs

The Class class

- To find out about a class, first get its Class object
  - If you have an object obj, you can get its class object with
    `Class c = obj.getClass();`
  - You can get the class object for the superclass of a Class c with
    `Class sup = c.getSuperclass();`
  - If you know the name of a class (say, Button) at compile time, you can get its class object with
    `Class c = Button.class;`
  - If you know the name of a class at run time (in a String variable str), you can get its class object with
    `Class c = class.forName(str);`

Getting the class name

- If you have a class object c, you can get the name of the class with c.getName()
- getName returns the fully qualified name; that is,

```
Class c = Button.class;
String s = c.getName();
System.out.println(s);
```

  will print

```
java.awt.Button
```

- Class Class and its methods are in java.lang, which is always imported and available

Getting all the superclasses

- getSuperclass() returns a Class object (or null if you call it on Object, which has no superclass)
- The following code is from the Sun tutorial:

```
static void printSuperclasses(Object o) {
    Class subclass = o.getClass();
    Class superclass = subclass.getSuperclass();
    while (superclass != null) {
        String className = superclass.getName();
        System.out.println(className);
        subclass = superclass;
        superclass = subclass.getSuperclass();
    }
}
```

Getting the class modifiers I

- The modifiers (e.g., public, final, abstract etc.) of a Class object is encoded in an int and can be queried by the method getModifiers().
- To decode the int result, we need methods of the Modifier class, which is in java.lang.reflect, so:

```
    import java.lang.reflect.*;
```

- Then we can do things like:

```
if (Modifier.isPublic(m))
    System.out.println("public");
```

Getting the class modifiers II

- Modifier contains these methods (among others):
  - public static boolean isAbstract(*int*)
  - public static boolean isFinal(*int*)
  - public static boolean isInterface(*int*)
  - public static boolean isPrivate(*int*)
  - public static boolean isProtected(*int*)
  - public static boolean isPublic(*int*)
  - public static String toString(*int*)
    - This will return a string such as

      "public final synchronized strictfp"

Getting interfaces

- A class can implement zero or more interfaces
- getInterfaces() returns an *array* of Class objects
- Ex:

```
  static void printInterfaceNames(Object o) {
      Class c = o.getClass();
      Class[] theInterfaces = c.getInterfaces();
      for (Class inf: interfaces) {
        System.out.println(inf.getName());     }}
```

- Note the convenience of enhanced for-loop

Examining classes and interfaces

- The class Class represents both classes and interfaces
- To determine if a given Class object c is an interface, use c.isInterface()
- To find out more about a class object, use:
  - getModifiers()
  - getFields() // "fields" == "instance variables"
  - getConstructors()
  - getMethods()
  - isArray()

Getting Fields

- public Field[] getFields() throws SecurityException
  - Returns an array of *public* Fields (including inherited fields).
  - The length of the array may be zero
  - The fields are not returned in any particular order
  - Both locally defined and inherited instance variables are returned, but *not* static variables.
- public Field getField(String name)

  throws NoSuchFieldException, SecurityException
  - Returns the named *public* Field
  - If no immediate field is found, the superclasses and interfaces are searched recursively

Using Fields, I

- If *f* is a Field object, then
  - *f*.getName() returns the simple name of the field
  - *f*.getType() returns the type (Class) of the field
  - *f*.getModifiers() returns the Modifiers of the field
  - *f*.toString() returns a String containing access modifiers, the type, and the fully qualified field name
    - Example: public java.lang.String Person.name
  - *f*.getDeclaringClass() returns the Class in which this field is declared
    - note: getFields() may return superclass fields.

Using Fields, II

- The fields of a particular object *obj* may be accessed with:
  - boolean *f*.getBoolean(*obj*), int *f*.getInt(*obj*), double *f*.getDouble(*obj*), etc., return the value of the field, assuming it is that type or can be widened to that type
  - Object *f*.get(obj) returns the value of the field, assuming it is an Object
  - void *f*.set(*obj*, *value*), void *f*.setBoolean(*obj*, *bool*), void *f*.setInt(*obj*, *i*), void *f*.getDouble(*obj*, *d*), etc. set the value of a field

Getting Constructors of a class

- if c is a Class, then
- c.getConstructors() : Constructor[] return an array of all public constructors of class c.
- c.getConstructor(Class … paramTypes) returns a constructor whose parameter types match those given paramTypes.

Ex:

- String.class.getConstructors().length

> 15;

- String.class.getConstrucor(char[].class, int.class, int.class).toString()

> String(char[], int,int)

Constructors

- If *c* is a Constructor object, then
  - c.getName() returns the name of the constructor, as a String (this is the same as the name of the class)
  - c.getDeclaringClass() returns the Class in which this constructor is declared
  - c.getModifiers() returns the Modifiers of the constructor
  - c.getParameterTypes() returns an array of Class objects, in declaration order
  - c.newInstance(Object... initargs) creates and returns a new instance of class *c*
    - Arguments that should be primitives are automatically unwrapped as needed

Example

- Constructor c = String.class.getConstrucor(char[].class, int.class, int.class).toString()

➢ String(char[], int,int).

- String s = c.newInstance(

  new char[] {‘a’,’b’,’c’,’d’ }, 1, 2);

- assert s == “bc”;

Methods

- public Method[] getMethods()

  throws SecurityException

  - Returns an array of Method objects
  - These are the *public member* methods of the class or interface, including inherited methods
  - The methods are returned in no particular order

- public Method getMethod(String name,

  Class... parameterTypes)

  throws NoSuchMethodException, SecurityException

Method methods, I

- getDeclaringClass()
  - Returns the Class object representing the class or interface that declares the method represented by this Method object
- getName()
  - Returns the name of the method represented by this Method object, as a String
- getModifiers()
  - Returns the Java language modifiers for the method represented by this Method object, as an integer
- getParameterTypes()
  - Returns an array of Class objects that represent the formal parameter types, in declaration order, of the method represented by this Method object

Method methods, II

- getReturnType()
  - Returns a Class object that represents the formal return type of the method represented by this Method object
- toString()
  - Returns a String describing this Method (typically pretty long)
- public Object invoke(Object obj, Object... args)
  - Invokes the underlying method represented by this Method object, on the specified object with the specified parameters
  - Individual parameters are automatically unwrapped to match primitive formal parameters

Examples of invoke()

- “abcdefg”.length()

> 7

- Method lengthMethod = String.class.getMethod(“length”) ;
- lengthMethod.invoke(“abcdefg”)

> 7

- “abcdefg”.substring(2, 5)

> cde

- Method substringMethod = String.class.getMethod (“substring”, int.class, Integer.TYPE) ;
- substringEMthod.invoke(“abcdefg”, 2, new Integer(5))

> cde

- To determine whether an object obj is an array,
  - Get its class c with Class c = obj.getClass();
  - Test with c.isArray()
- To find the type of components of the array,
  - c.getComponentType()
    - Returns null if c is not the class of an array
- Ex:
  - int[].class.isArray() == true ;
  - int[].class.getComponentType() == int.class

Arrays II

- The Array class in java.lang.reflect provides *static* methods for working with arrays
- To create an array,
- Array.newInstance(Class componentType, int size)
  - This returns, as an Object, the newly created array
    - You can cast it to the desired type if you like
  - The componentType may itself be an array
    - This would create a multiple-dimensioned array
    - The limit on the number of dimensions is usually 255
- Array.newInstance(Class componentType, int... sizes)
  - This returns, as an Object, the newly created multidimensional array (with sizes.length dimensions)

Examples

- The following two objects are of the same type:
  - new String[10]
  - Array.newInstance(String.class, 10)
- The following two objects are of the same type:
  - new String[10][20]
  - Array.newInstance(String.class, 10, 20)

Arrays III

- To get the value of array elements,
  - Array.get(Object array, int index) returns an Object
  - Array.getBoolean(Object array, int index) returns a boolean
  - Array.getByte(Object array, int index) returns a byte
  - etc.
- To store values into an array,
  - Array.set(Object array, int index, Object value)
  - Array.setInt(Object array, int index, int i)
  - Array.setFloat(Object array, int index, float f)
  - etc.

Examples

- a = new int[] {1,2,3,4};
- Array.getInt(a, 2)   // → 3
- Array.setInt(a, 3, 5) // a = {1,2,3, 5 }.

- s = new String[] { “ab”, “bc”, “cd” };
- Array.get(s, 1) // → “bc”
- Array.set(s, 1, “xxx”) // s[1] = “xxx”

Getting non-public members of a class

- All getXXX() methods of Class mentioned above return only public members of the target (as well as ancestor) classes, but they cannot return non-public members.
- There are another set of getDeclaredXXX() methods in Class that will return all (even private or static) members of target class but no inherited members are included.
- getDeclaredConstructors(), defDeclaredConstrucor(Class…)
- getDeclaredFields(),
  getDeclaredField(String)
- getDeclaredmethods(),
  getDeclaredMethod(String, Class…)

Example

- String.class.getConstructors().length

> 15

- String.class.getDeclaredConstructors().length

> 16.

- Constructor[] cs = String.class.getDeclaredConstructors();

```
for(Constructor c : cs)
   if( ! (Modifier.isPublic(c.getModifiers())))
      out.println(c);
```

> java.lang.String(int,int,char[]) // package

Concluding comments

- Many of these methods throw exceptions not described here
  - For details, see the Java API
- Reflection isn’t used in “normal” programs, but when you need it, it’s indispensable
- Studying the java reflection package gives you a chance to review the basics of java class structure.