



Ch1. Java Review

© copyright 2006 SNU IDB Lab



Bird's eye view

- Requirements of program development
 - Representing data in an effective way
 - We need **data structures**
 - Developing a suitable procedure
 - We need **algorithm design methods**
- Before you go, you need to be
 - A proficient java programmer
 - An adept analyst of computer programs

adept = skillful = proficient



Table of Contents

- **Introduction**
- **Structure of a Java Program**
- **The Java Compiler and Virtual Machine**
- **Documentation Comments**
- **Data Types & Methods**
- **Exceptions**
- **Your Very Own Data Type**
- **Access Modifiers**
- **Inheritance and Method Overriding**
- **Defining an Exception Class**
- **Generic Methods**
- **Garbage Collection**
- **Recursion**
- **Testing and Debugging**



Introduction (1/2)

- When examining a computer program
 - Is it **correct**?
 - How **easy** is it to read the program?
 - Is the program **well documented**?
 - How easy is it to **make changes** to the program?
 - **How much memory** is needed to run the program?
 - For **how long** will the program run?
 - **How general** is the code?
 - Can the code be compiled & run on **a variety of computers**?



Introduction (2/2)

- Regardless of the application, the most important attribute of a program is **correctness**
- The goal of software courses is to teach techniques that will enable you to develop **correct, elegant, and efficient programming**
- This course “data structure” is regarding **efficient programming techniques**
- Let’s begin with JAVA



Table of Contents

- **Introduction**
- **Structure of a Java Program**
- **The Java Compiler and Virtual Machine**
- **Documentation Comments**
- **Data Types & Methods**
- **Exceptions**
- **Your Very Own Data Type**
- **Access Modifiers**
- **Inheritance and Method Overriding**
- **Defining an Exception Class**
- **Generic Methods**
- **Garbage Collection**
- **Recursion**
- **Testing and Debugging**



Java Skeleton (1)

- Every Java program
 - Class with data member & method member
- Stand-alone program
 - Method *main()*
 - *javac programname, and then java programname*
- Applet
 - After compilation, applet is embedded in HTML
 - Method *init()*
 - Web browser or applet viewer



Java Skeleton (2)

- Source File
 - A plain text file (`*.java`) containing the Java code
 - The Java compiler (`javac`) processes the source file to produce a byte code file (`*.class`)
 - A source file may have
 - Only one public class (or interface)
 - And an unlimited # of default classes (or interfaces) defined within it
 - The name of the source file must be the same as the name of the “only one” public class
 - Package name can be specified in the source file
 - Compilation created the directory for the classes in a package



Java Skeleton (3)

- **Declarations**

- A declaration introduces a class, interface, method, package, or variable into a Java program
- The order in which you place your declarations in the source file is important
 - **1. The package declaration (optional, at the top)**
 - **2. The import function (optional)**
 - **3. Any class declarations**

- **Class**

- A declaration for a potential object
- You can think of a class as a skeleton or framework that contains methods but no data
- Once the class is initiated it becomes *an object (or an instance)*



Java Skeleton (4)

- Package
 - An entity that groups classes together
 - Packages perform the following functions
 - Organize and group classes
 - Help to specify access restrictions to variables and methods
 - Make it easy to import other programmers' classes into your code
 - The name of the package must reflect **the directory structure** used to store the classes in your package after compilation
 - Place the package declaration at the top of the source file



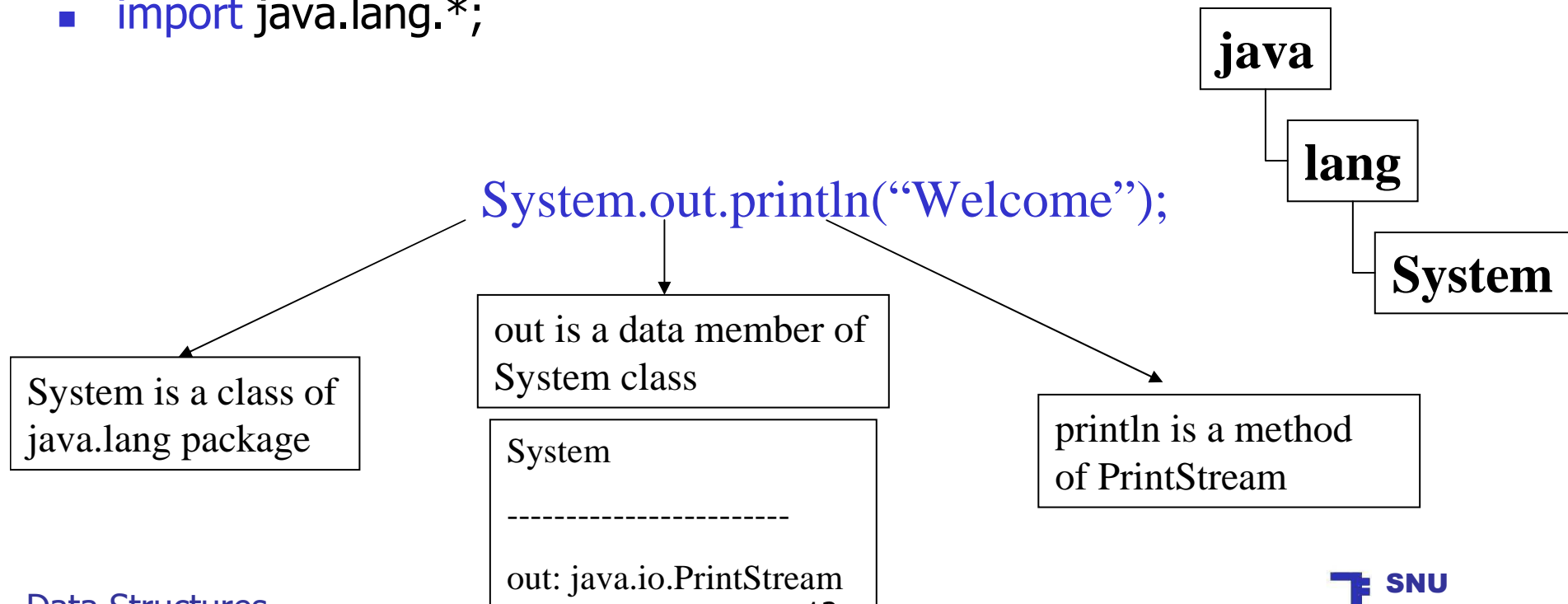
Java Skeleton (5)

- Each package defines a directory at compile time

```
package misc; //This program is a member of the package misc
public class Welcome //name of this class is Welcome
{
    public static void main(String args[])
    {
        System.out.println("Welcome to the text Data Structures");
    }
} //Welcome.java is in the directory misc
```

Importing Classes and Packages (1/2)

- An **import** statement allows you to use shorthand in your source code so that we don't have to type a fully qualified name for each class you use
 - Use a class that is contained in some other program
 - Require a path to this class
- **import** java.lang.*;



Importing Classes and Packages (2/2)

- You can select classes within a package, or select an entire package of classes that you may use in your code

```
package misc;
```

```
import java.lang.*;           //default (imported automatically)
```

```
import java.io.*;           //import entire java.io package
```

```
import java.io.PrintStream; //import java.io.PrintStream class
```

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

```
class myclass { ..... }
```

```
class myextraclass { ..... }
```

If you declare `java.io.*`,
`java.io.PrintStream` is
not necessary

Superclasses and Subclasses (1/2)

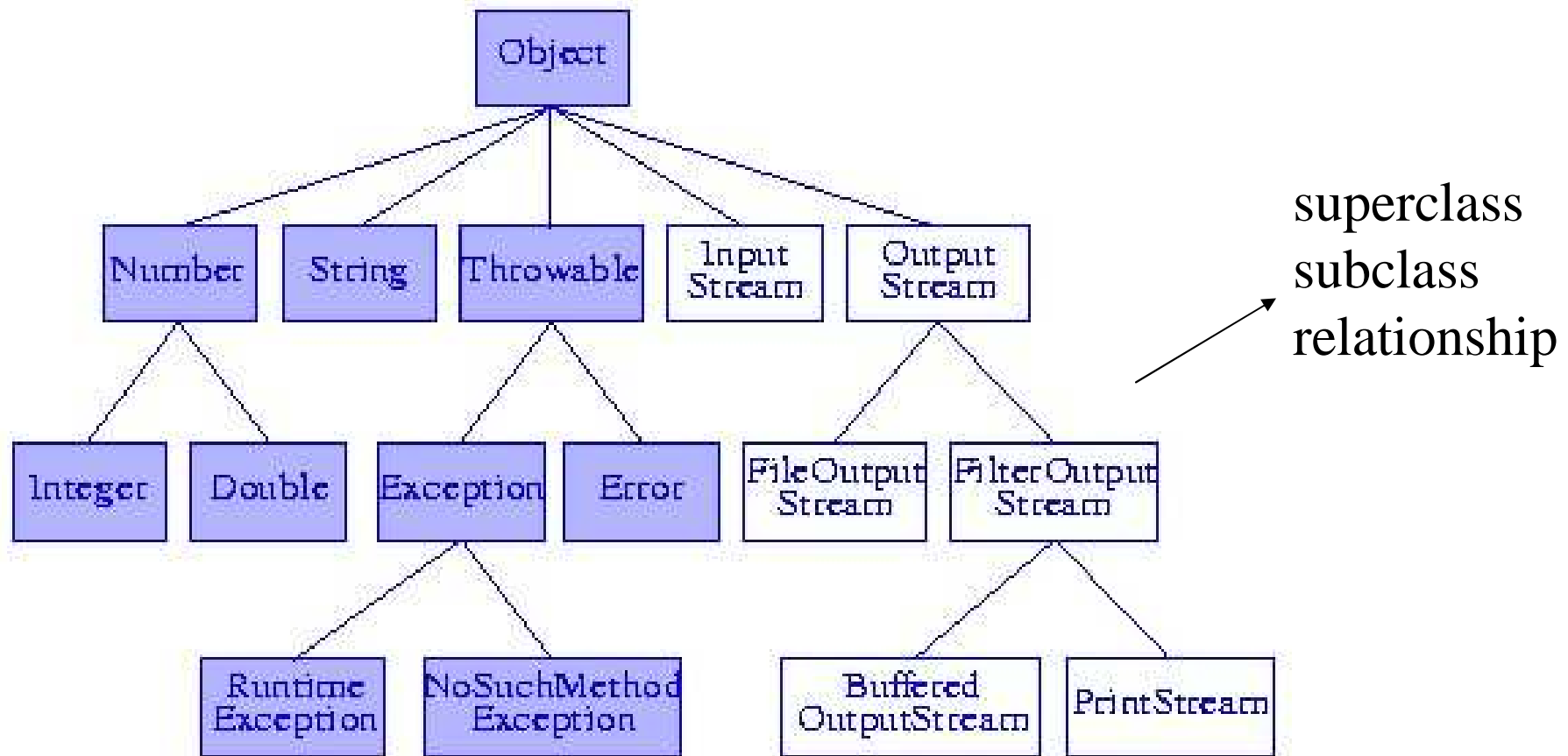
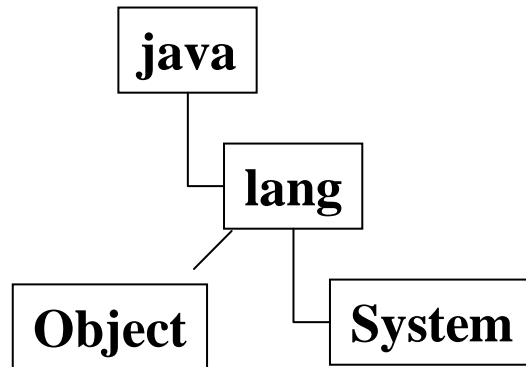


Figure 1.1 Hierarchy for a few Java classes

Superclasses and Subclasses (2/2)

- All classes are subclasses of `java.lang.Object`
- No class has more than one superclass: **single inheritance**



```
public class Welcome extends NameOfSuperClass // specify the superclass
```

```
public final class Welcome // prevent extending the class
```

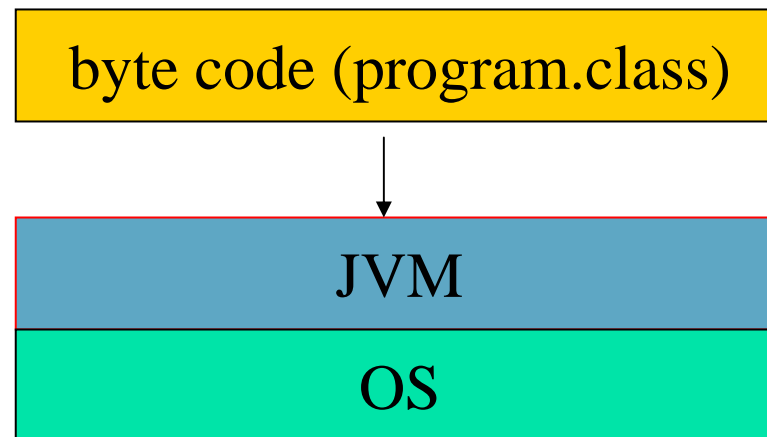


Table of Contents

- **Introduction**
- **Structure of a Java Program**
- **The Java Compiler and Virtual Machine**
- **Documentation Comments**
- **Data Types & Methods**
- **Exceptions**
- **Your Very Own Data Type**
- **Access Modifiers**
- **Inheritance and Method Overriding**
- **Defining an Exception Class**
- **Generic Methods**
- **Garbage Collection**
- **Recursion**
- **Testing and Debugging**

The JAVA Compiler and Virtual Machine (1/2)

- JAVA compiler
 - `Javac ProgramName.java` // generates `ProgramName.class`
 - `Java ProgramName` // JVM interprets `ProgramName.class`





The JAVA Compiler and Virtual Machine (2/2)

- JVM Advantage
 - Write Once, Run Anywhere → Portability
 - Compact size
 - Higher level of security
- JVM Disadvantage
 - Slower than C or C++



Table of Contents

- **Introduction**
- **Structure of a Java Program**
- **The Java Compiler and Virtual Machine**
- **Documentation Comments**
- **Data Types & Methods**
- **Exceptions**
- **Your Very Own Data Type**
- **Access Modifiers**
- **Inheritance and Method Overriding**
- **Defining an Exception Class**
- **Generic Methods**
- **Garbage Collection**
- **Recursion**
- **Testing and Debugging**



Documentation Comments (1/3)

- Three ways to write comments
 - `//` beginning with a double slash
 - `/*` and ending with `*/` → general comments
 - `/**` and ending with `*/` → documentation comments
 - Java documentation generation system “javadoc”

`javadoc -d docDirectory nameOfPackage`



generation of documentation in the
docDirectory for nameOfPackage



Documentation Comments (2/3)

```
/** Method to find the position of the largest integer.  
@param a is the array containing the integers  
@param n gives the position of the last integer  
@throws IllegalArgumentException when n < 0  
@return position of max element in a[0:n] */
```

```
public static int max(int [] a, int n)  
{  
    implementation of max comes here  
}
```




Documentation
source



Documentation Comments (3/3)

```
javadoc -d docDirectory nameOfPackage
```



Documentation
result

```
Public static int max(int a[], int n)
```

Method to find the position of the largest integer.

Parameters:

a is the array containing the integers

n gives the position of the last integer

Returns:

position of max element in a[0:n]

Throws:

IllegalArgumentException - when $n < 0$



Table of Contents

- **Introduction**
- **Structure of a Java Program**
- **The Java Compiler and Virtual Machine**
- **Documentation Comments**
- **Data Types & Methods**
- **Exceptions**
- **Your Very Own Data Type**
- **Access Modifiers**
- **Inheritance and Method Overriding**
- **Defining an Exception Class**
- **Generic Methods**
- **Garbage Collection**
- **Recursion**
- **Testing and Debugging**



Data Types (1/3)

Type	Default	Space (bits)	Range
boolean	false	1	[true, false]
byte	0	8	[-128, 127]
char	\u0000	16	[\u0000, \uFFFF]
double	0.0	64	$\pm 4.9 * 10^{-324}$ to $\pm 1.8 * 10^{308}$
float	0.0	32	$\pm 1.4 * 10^{-45}$ to $\pm 3.4 * 10^{38}$
int	0	32	-2,147,483,648 to 2,147,483,647
long	0	64	$\pm 9.2 * 10^{17}$
short	0	16	[-32768, 32767]

Java's primitive data types



Data Types (2/3)

- Primitive Data Types
 - int, float, long, short, boolean, byte, char, double
- Nonprimitive Data Types
 - Byte, Integer, Boolean, String
 - Declared in java.lang package
 - They have many useful methods of their own

```
String s = "hello";  
System.out.println ( "The length of s is " + s.length() );
```



Data Types (3/3)

- Creation of an object instance
 - Primitive data type
 - `int theInt;` // an instance is created & default value is assigned
 - Nonprimitive data type
 - `String s;` // it creates an object that can reference a string
 - `String s = new String("Bye");`



Methods (1/2)

- Method

- A function or procedure to perform a well-defined task

```
public static int abc(int a, int b, int c)           //formal parameters
{ return a+b*c+b/c;
}
... ..
z = abc(2, x, y)                                     // actual parameters
```

- In Java

- All method parameters are **value parameters**



Methods (2/2)

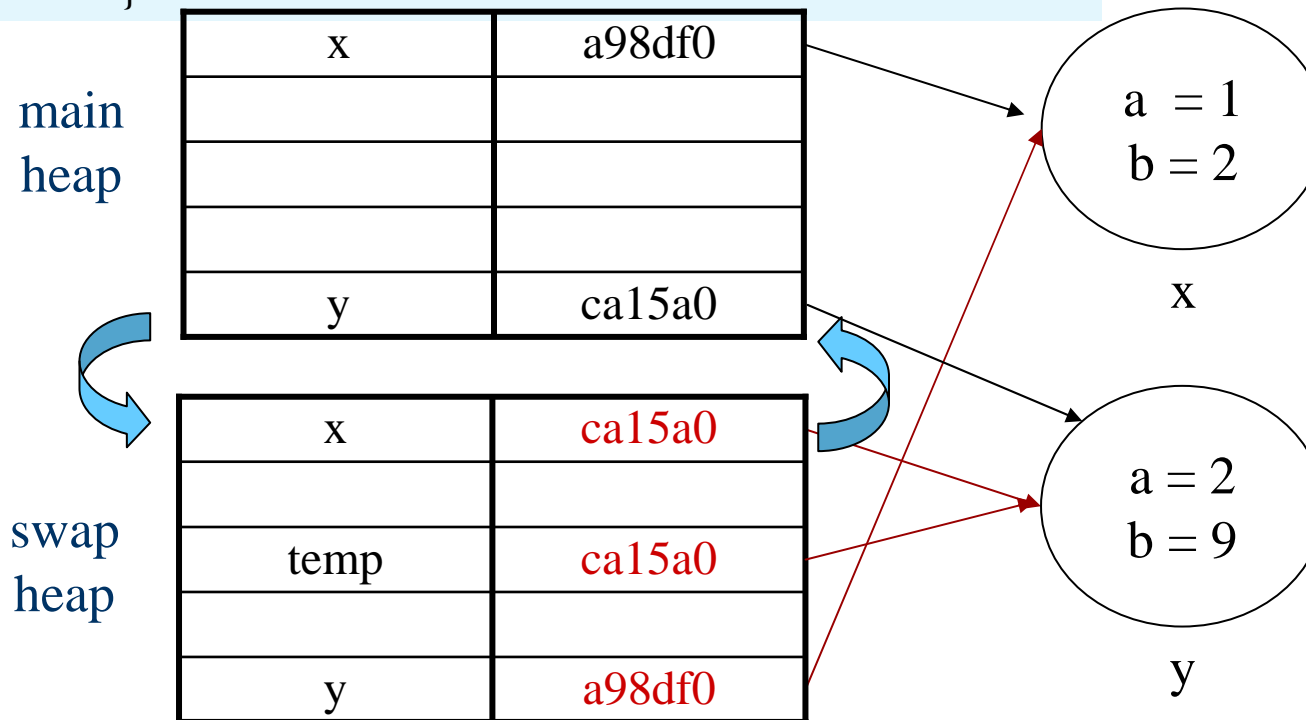
- Java allows **method overloading**
 - Same name with different signature

```
public static int abc(int a, int b, int c) {  
    return a+b*c+b/c;  
}  
  
public static float abc(float a, float b, float c) {  
    return a+b*c+b/c;  
}  
  
... ..  
z = abc(10, 11, 12);  
z = abc(9.9, 10.0, 10.1);
```

SWAP – Changing References

```
public static void swap (Example x, Example y){  
    Example temp = x ;  
    x = y ;  
    y = temp ;  
}
```

```
public class Example { int a; int b }
```



SWAP – Changing Data Member

```
public static void swap2 (Example x, Example y){  
    int temp = x.a;  
    x.a     = y.a;  
    y.a     = temp;  
}
```

```
public class Example { int a; int b }
```

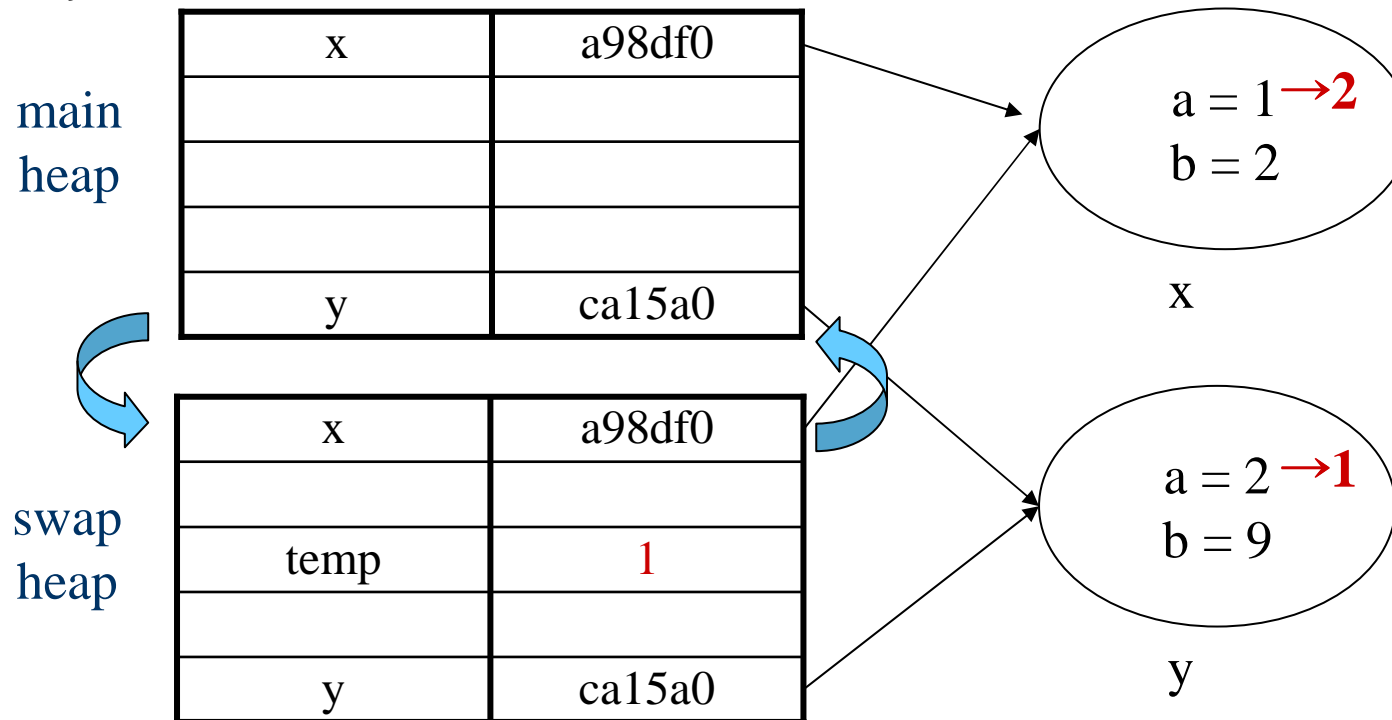




Table of Contents

- **Introduction**
- **Structure of a Java Program**
- **The Java Compiler and Virtual Machine**
- **Documentation Comments**
- **Data Types & Methods**
- **Exceptions**
- **Your Very Own Data Type**
- **Access Modifiers**
- **Inheritance and Method Overriding**
- **Defining an Exception Class**
- **Generic Methods**
- **Garbage Collection**
- **Recursion**
- **Testing and Debugging**



Exceptions (1/4)

- Exception
 - Means “exceptional condition”
 - an occurrence that alters the normal program flow
 - Caused by a variety of happenings
 - HW failures
 - Resource exhaustion
 - Good old bugs
 - When an exceptional event occurs,
 - An exception is said to be *thrown*
 - The code that is responsible for doing something about the error is called *an exception handler*
 - The exception handler *catches* the exception



Exceptions (2/4)

- Throwing an Exception
 - System Built Exceptions
 - *ArithmeticException, ArrayIndexOutOfBoundsException, IllegalArgumentException, IOException, RuntimeException, Error,.....*
 - Can be taken care of by system **automatically**
 - Some system built exceptions can be thrown by the user's program
 - *public static int abc(int a, int b) throws IllegalArgumentException*
 - Proper **exception handler** must be provided by the user
 - Some system built exceptions cannot be thrown by the user's program
 - subclasses of either *RuntimeException* class or *Error class*



Exceptions (3/4)

- Handling Exceptions
 - `try-catch-finally` block
 - In a try block, exceptions can occur
 - In a catch block, exceptions are handled
 - Codes in finally block always get executed

```
public static int getVolume (int a, int b, int c) {  
    if (a <= 0 || b <= 0 || c <= 0)  
        throw new IllegalArgumentException ("All parameters must be > 0");  
    else  
        return a * b * c;  
}
```



Exceptions (4/4)

```
public static void main(String [] args) {
    try {
        System.out.println ( getVolume(2, -3, 4) );
    }
    catch (IllegalArgumentException e) {
        System.out.println ("Some parameters have minus values.");
        System.out.println (e);
    }
    catch (Throwable e) { // Throwable class → Exception Class → many exceptions
        System.out.println (e);
    }
    finally { // this code gets executed whether or not an exception is thrown in the try block
        System.out.println("Thanks for trying this program");
    }
}
```



Table of Contents

- **Introduction**
- **Structure of a Java Program**
- **The Java Compiler and Virtual Machine**
- **Documentation Comments**
- **Data Types & Methods**
- **Exceptions**
- **Your Very Own Data Type**
- **Access Modifiers**
- **Inheritance and Method Overriding**
- **Defining an Exception Class**
- **Generic Methods**
- **Garbage Collection**
- **Recursion**
- **Testing and Debugging**



Your Very Own Data Type

- There is a case that you want to specify the format of U.S money with a sign, dollars and cents and avoid the situation that
 - dollars < 0
 - cents < 0 or cents > 99
- If you declare the currency simply with [the float data type](#), you cannot avoid the situation such as
 - Currency = 35.4755
 - Currency = -9.888



The Class *Currency*

- Define your own data types : *Currency* class
- Components of *Currency* : *Data Member*
 - sign, dollars, cents
- Operations for *Currency* : *Methods*
 - Set their value
 - Determine the components
 - Add two instances of type *Currency*
 - Increment the value of an instance
 - Output

```
public class Currency
{
    // data and method members of Currency come here
}
```



The Data Members of *Currency*

- Five data members
 - PLUS, MINUS, sign, dollars, cents
- Keywords
 - Public : Visible to all classes
 - Static : PLUS and MINUS are class data members
 - Final : PLUS and MINUS cannot be changed
 - Private : Visible only within the class *Currency*

```
// class constants
public static final boolean PLUS = true;
public static final boolean MINUS = false;
// instance data members
private boolean sign;
private long dollars;
private byte cents;
```



The Method Members of *Currency*

- Constructor methods
 - Automatically invoked when an instance is created
- Accessor methods
 - Return the value of data member
- Mutator methods
 - Change the data member value
- Output methods
 - Converting a class instance into its output format
- Arithmetic methods
 - Arithmetic operations on class instances
- Main method
 - Present in all stand-alone Java applications



The Constructors of *Currency* (1)

- Constructor Method name
 - Same as the class name
- Access modifier: default, public, protected, private
 - `public` `Currency()`
- Initialize the data members

```
public Currency (boolean theSign, long theDollars, byte theCents)
```

- `this()` invokes a constructor with the same signature
- Constructors never return a value



The Constructors of *Currency* (2)

```
/** initialize instance to theSign $ theDollars.theCents
 @throws IllegalArgumentException when theDollars < 0 or theCents < 0 or theCents > 99 */
public Currency(boolean theSign, long theDollars, byte theCents)
{ sign = theSign;
  if (theDollars < 0)
    throw new IllegalArgumentException ("Dollar value must be >= 0");
  else dollars = theDollars;
  if (theCents < 0 || theCents > 99)
    throw new IllegalArgumentException ("Cents must be between 0 and 99");
  else cents = theCents;
}

public Currency() /** initialize instance to $0.00 */
  {this(PLUS, 0L, (byte) 0);}
public Currency(double theValue) /** initialize with double */
  {setValue(theValue);}
```



Creating Instances of *Currency*

```
Currency g, h, i, j; //declare variables  
g = new Currency(); //create instances using constructors  
h = new Currency(PLUS, 3L, (byte)50);  
i = new Currency(-2.50);  
j = new Currency();
```

Or

```
Currency g = new Currency();
```



The Accessor Methods of *Currency*

```
/** @return sign */  
public boolean getSign()  
    {return sign;}
```

```
/** @return dollars */  
public long getDollars()  
    {return dollars;}
```

return the value
of an object

```
/** @return cents */  
public byte getCents()  
    {return cents;}
```



The Mutator Methods of *Currency* (1)

- Set or change the characteristics of an object

```
/** set sign = theSign */
```

```
public void setSign(boolean theSign)  
    { sign = theSign; }
```

```
/** set dollars = theDollars
```

```
 * @throws IllegalArgumentException when theDollars < 0 */
```

```
public void setDollars(long theDollars){  
    if (theDollars < 0)  
        throw new IllegalArgumentException ("Dollar value must be >= 0");  
    else    dollars = theDollars;  
}
```



The Mutator Methods of *Currency* (2)

```
/** set sign, dollars, and cents from a “double” data */
```

```
public void setValue(double theValue) {  
    if (theValue < 0) {  
        sign = MINUS;  
        theValue = - theValue; }  
    else sign = PLUS;  
    dollars = (long) (theValue + 0.005); // extract integral part  
    cents = (byte) ((theValue + 0.005 - dollars) * 100); // get two decimal digits  
}
```

```
/** set sign, dollars, and cents from a “Currency” object*/
```

```
public void setValue(Currency x)  
{ sign = x.sign;  
  dollars = x.dollars;  
  cents = x.cents;
```

Invoking Methods & Accessing Data Members



- Instance methods
 - `byte gCents = g.getCents()`
- Class methods
 - `double a = Math.sqrt(4.0)`
- Accessing data members
 - `byte cents = x.cents`



Output Method for *Currency*

- The output method: `toString()`
 - Defined in the *Object* class
 - Redefine `toString()`
 - We can get the Java to output our object

```
/** convert to a string */  
public String toString()  
{ if (sign == PLUS) {return "$" + dollars + "." + cents;}  
  else {return "-$" + dollars + "." + cents;}  
}
```




Arithmetic Methods for *Currency*

```
public Currency add(Currency x) /** @return this + x */
{
    long a1 = dollars * 100 + cents; // convert this to a long
    if (sign == MINUS)    a1 = -a1;
    long a2 = x.dollars * 100 + x.cents; // convert x to a long
    if (x.sign == MINUS)  a2 = -a2;
    long a3 = a1 + a2;
    answer = new Currency(); // convert result to Currency object
    if (a3 < 0) { answer.sign = MINUS;    a3 = -a3; }
    else    answer.sign = PLUS;
    answer.dollars = a3 / 100;
    answer.cents = (byte) (a3 - answer.dollars * 100);
    return answer;
}

public Currency increment(Currency x) /** @return this incremented by x */
{
    setValue(add(x));
    return this;
}
```



Main method for *Currency* (1)

```
public static void main(String [] args)
{ // test constructors
    Currency g = new Currency(),    h = new Currency(PLUS, 3L, (byte) 50),
      i = new Currency(-2.50), j = new Currency();

    // test toString
    System.out.println ("The initial values are: " + g + " " + h + " " + i + " " + j );
    System.out.println();

    // test mutators;  first make g nonzero
    g.setDollars(2);
    g.setSign(MINUS);
    g.setCents((byte) 25);
    i.setValue(-6.45);
    System.out.println ("New values are " + g + " " + i);
    System.out.println();
```



Main method for *Currency* (2)

```
// do some arithmetic
```

```
    j = h.add(g);
```

```
    System.out.println (h + " + " + g + " = " + j);
```

```
System.out.print (i + " incremented by " + h + " is ");
```

```
    i.increment(h);
```

```
    System.out.println(i);
```

```
    j = i.add(g).add(h);
```

```
    System.out.println (i + " + " + g + " + " + h + " = " + j);
```

```
    System.out.println();
```

```
    j = i.increment(g).add(h);
```

```
    System.out.println(j);
```

```
    System.out.println(i);
```

```
}
```



Table of Contents

- **Introduction**
- **Structure of a Java Program**
- **The Java Compiler and Virtual Machine**
- **Documentation Comments**
- **Data Types & Methods**
- **Exceptions**
- **Your Very Own Data Type**
- **Access Modifiers**
- **Inheritance and Method Overriding**
- **Defining an Exception Class**
- **Generic Methods**
- **Garbage Collection**
- **Recursion**
- **Testing and Debugging**



Access Modifiers

<u>Access Modifier</u>	<u>Member Visibility</u>
default	member is visible only to classes in the same package
private	member is visible only within the class C
protected	member is visible to all classes in the same package and to subclasses of C in other packages
public	member is visible to all classes in all packages



Table of Contents

- **Introduction**
- **Structure of a Java Program**
- **The Java Compiler and Virtual Machine**
- **Documentation Comments**
- **Data Types & Methods**
- **Exceptions**
- **Your Very Own Data Type**
- **Access Modifiers**
- **Inheritance and Method Overriding**
- **Defining an Exception Class**
- **Generic Methods**
- **Garbage Collection**
- **Recursion**
- **Testing and Debugging**



Inheritance and Method Overriding

- Inheritance
 - Data and method member from the superclass
 - Newly defined member for the new class
 - “extends” means “ISA” relationship
- Method overriding
 - Same signature as superclass
 - Newly defined method is invoked
- No more overriding
 - `public final boolean equals(Object theObject)`
 - static, private methods cannot be overridden either



Currency Revisited (1)

- *Currency* has many data members, which make the class complicated
- ➔ What if we use only one data member named '*amount*' which is cents representation of the given money?

Money	Data members of <i>Currency</i>	Data members of <i>CurrencyAsLong</i>
\$1.32	sign=PLUS dollars=1 cents=32	amount=132
-\$0.20	sign=MINUS dollars=0 cents=20	amount=-20



Currency Revisited (2)

CurrencyAsLong:

- Currency with additional private data member “amount” which is of type long

```
public class CurrencyAsLong extends Currency {  
    long amount;
```

```
    public CurrencyAsLong add(CurrencyAsLong x) /** @return this + x */  
        {return new CurrencyAsLong(amount + x.amount);}
```

```
    public CurrencyAsLong increment(CurrencyAsLong x)  
        /** @return this incremented by x */  
        { amount += x.amount;  
          return this;  
        }
```



Currency Revisited (3)

```
public boolean getSign() { /** @return sign */
    if (amount < 0) return MINUS;
    else return PLUS;
}
```

```
public long getDollars(){ /** @return dollars */
    if (amount < 0) return - amount / 100;
    else return amount / 100;
}
```

```
/** Set the sign of amount to theSign.*For this to work properly amount must be nonzero. */
```

```
public void setSign(boolean theSign){
    // change the sign as necessary
    if ((amount < 0 && theSign == PLUS) || (amount > 0 && theSign == MINUS))
        amount = -amount;
}
} // end of class CurrencyAsLong
```



Table of Contents

- **Introduction**
- **Structure of a Java Program**
- **The Java Compiler and Virtual Machine**
- **Documentation Comments**
- **Data Types & Methods**
- **Exceptions**
- **Your Very Own Data Type**
- **Access Modifiers**
- **Inheritance and Method Overriding**
- **Defining an Exception Class**
- **Generic Methods**
- **Garbage Collection**
- **Recursion**
- **Testing and Debugging**



User-defined Exception Class

- We can define our own exception

```
public static void main(String[] args){
    int n = Integer.parseInt(args[0]);
    if (n == 0) throw new DivideByZeroException();
    System.out.println("Inverse of " + n + " is " + (1.0 / n));
}

class DivideByZeroException extends ArithmeticException{
    public DivideByZeroException() {
        System.out.println("Can not divide a number by zero!");
    }
}
```



Table of Contents

- **Introduction**
- **Structure of a Java Program**
- **The Java Compiler and Virtual Machine**
- **Documentation Comments**
- **Data Types & Methods**
- **Exceptions**
- **Your Very Own Data Type**
- **Access Modifiers**
- **Inheritance and Method Overriding**
- **Defining an Exception Class**
- **Generic Methods**
- **Garbage Collection**
- **Recursion**
- **Testing and Debugging**



Generic Methods (1/3)

- Similar methods differing only in the data types of the formal parameters

```
public void swap (int[] a, int i, int j){  
    int temp = a[i];  
    a[i] = a[j];  
    a[j] = temp;  
}
```

```
public void swap (float[] a, int i, int j){  
    float temp = a[i];  
    a[i] = a[j];  
    a[j] = temp;  
}
```

- One generic method can do the same job

```
public void swap(Object[] a, int i, int j){  
    Object temp = a[i];  
    a[i] = a[j];  
    a[j] = temp;  
}
```



Generic Methods (2/3)

- Primitive type → Impossible to write generic code

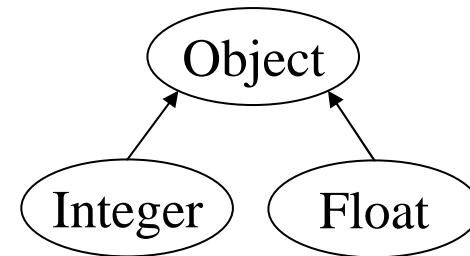
```
public static void swap (int [] a, int i, int j) {  
    // Don't bother to check that indexes i and j are in bounds.  
    // Java will do this and throw an ArrayIndexOutOfBoundsException  
    // if i or j is out of bounds.  
  
    int temp = a[i];  
    a[i] = a[j];  
    a[j] = temp;  
}
```

Generic Methods (3/3)

- Non primitive type → Generic code works for subclasses

```
public static void swap ( Object [] a, int i, int j){  
    Object temp = a[i];  
    a[i] = a[j];  
    a[j] = temp;  
}
```

```
public static void main ( String[] args){  
    Integer [] i = {new Integer(1), new Integer(3), new Integer(2)};  
    Float [] f = {new Float(0.3), new Float(0.2), new Float(0.1)};  
    swap ( i, 1, 2); // {1, 2, 3}  
    swap ( f, 0, 2); // {0.1, 0.2, 0.3}  
}
```

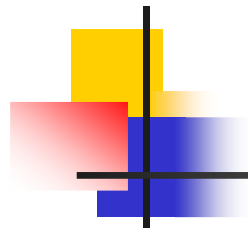




Interface and Generic Method

- A Java interface
 - A list of zero or more **static final** data members
 - A list of zero or more **method headers**
 - No implementation is provided
 - No instance is created

- Similar to C++ “Abstract Class”



Interface *Computable*

/ Interface to be implemented by all classes that permit the standard arithmetic operations. */**

```
public interface Computable{  
    public Object add(Object x) ;           /** @return this + x */  
    public Object subtract(Object x) ;      /** @return this - x */  
    public Object multiply(Object x) ;      /** @return this * x */  
    public Object divide(Object x) ;        /** @return quotient of this / x */  
    public Object mod(Object x) ;           /** @return remainder of this / x */  
    public Object increment(Object x) ;     /** @return this incremented by x */  
    public Object decrement(Object x) ;     /** @return this decremented by x */  
    public Object zero() ;                 /** @return the additive zero element */  
    public Object identity() ;             /** @return the multiplicative identity element */  
}
```



Generic method using *Comparable*

■ Using method overload

```
public static int abc (int a, int b, int c)    public static float abc (float a, float b, float c)
{
    return a + b*c + b/c;                    {
                                             return a + b*c + b/c;
    }                                         }
```

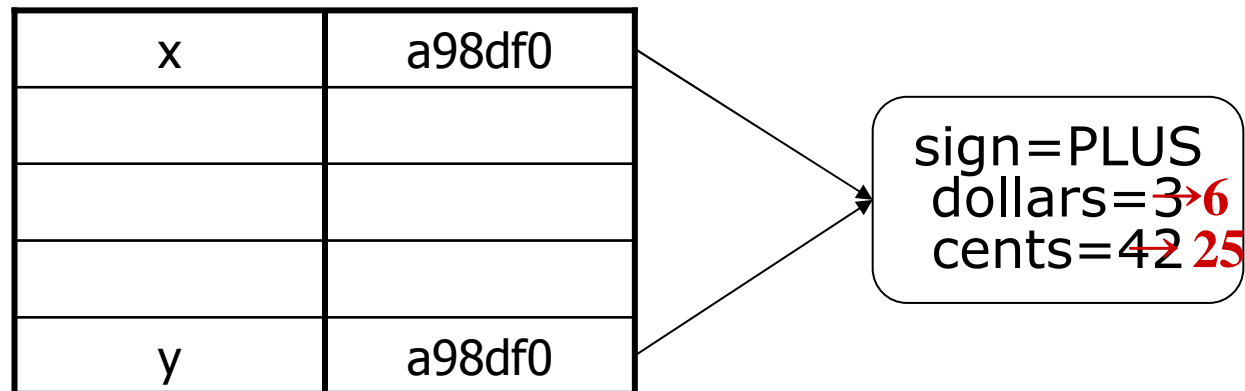
■ Using generic method

```
public static Comparable abc (Comparable a, Comparable b, Comparable c)
{
    Comparable t = (Comparable) a.add (b.multiply(c));
    return (Comparable) t.add (b.divide(c));
}
```

Copying a reference vs. Cloning an object (1)

- Copying a reference

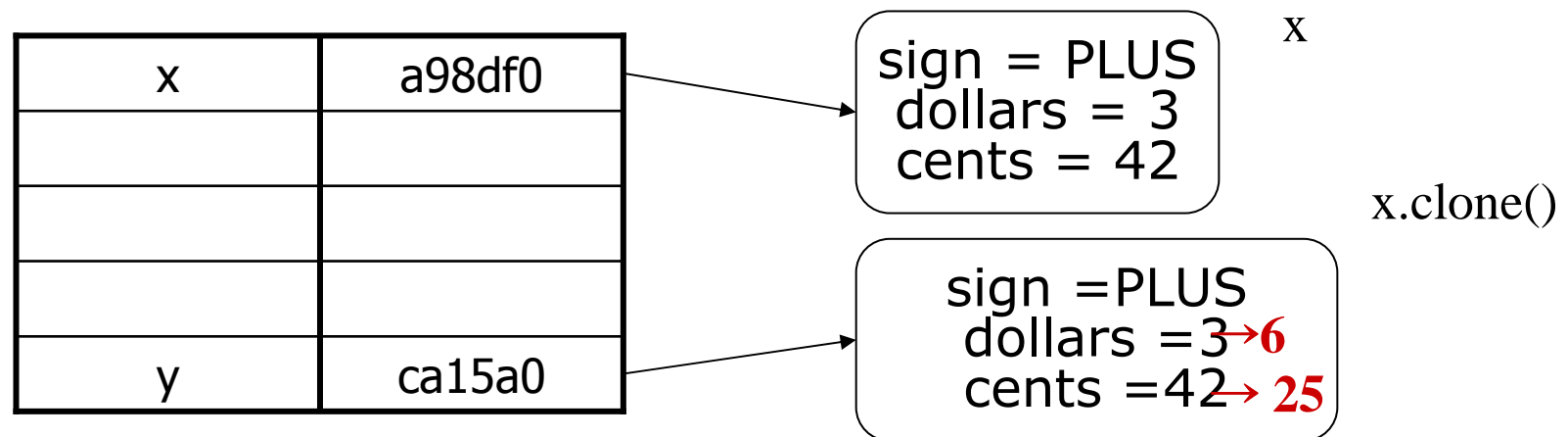
```
x = new Currency(3.42);  
y = x;  
y.setValue(6.25);  
System.out.println(x); // print "$6.25"
```



Copying a reference vs. Cloning an object (2)

- Cloning an object

```
x = new Currency(3.42);  
y = x.clone();  
y.setValue(6.25);  
System.out.println(x); // "$3.42"
```





Implementing Interface (1)

- public interface `Comparable` {...}
- public interface `Comparable` {...}
- public interface `Operable` extends `Comparable`, `Comparable` {}
- Public interface `Zero` { public int zero() }
- Public interface `CloneableObject` { public clone() }

- If a class “implements” one or more interfaces, **all of the methods** in the interfaces should be implemented



Implementing the Interface (2)

```
public class MyInteger implements Operable, Zero, CloneableObject {
    private int value; // value of the integer
    public MyInteger (int theValue) {value = theValue;} // constructor

    // a Computable interface method : @return this + x */
    public Object add(Object x)
    {return new MyInteger (value + ((MyInteger) x).value); }

    // Comparable interface method
    /** @return -1 if this < x, return 0 if this == x, return 1 if this > x */
    public int compareTo(Object x){
        int y = ((MyInteger) x).value;
        if (value < y) return -1;
        if (value == y) return 0;
        return 1;
    }
} // Only some method implementations are shown!
```



Finding out the types of parameters

- Example generic code (incomplete)

```
public void myFunc (Object obj) {  
    // How can we find out the type of obj ?  
    If obj is an instance of String, print “String”  
    If obj is an instance of Integer, print “Integer”  
}
```

- Every object has a method named `getClass()`, which returns the type of object
e.g. If x is a String, `x.getClass()` returns `String.class`

- Example generic code (complete)

```
public void myFunc (Object obj) {  
    if ( obj.getClass() == String.class) System.out.println(“String”);  
    if ( obj.getClass() == Integer.class) System.out.println(“Integer”);  
}
```




The method *inputArray*

```
/** input objects of type theClass and store into an array */
public void inputArray (Class theClass, MyInputStream stream)
{ try { // get the proper method to be used to read in the values
    Class [] parameterTypes = {MyInputStream.class};
    Method inputMethod = theClass.getMethod("input", parameterTypes);
    // input number of elements and create an array of that size
    System.out.println("Enter number of elements");
    int n = stream.readInteger();
    a = new Object [n];
    Object [] inputMethodArgs = {stream}; // input the elements
    for (int i = 0; i < n; i++) {
        System.out.println("Enter element " + (i+1));
        a[i] = inputMethod.invoke(null, inputMethodArgs); }
    } // end of try
catch (Exception e) { System.out.println(e);
    throw new IllegalArgumentException("Array1D.inputArray");
}
```



The alternative *inputArray*

```
/** input objects of type theClass and store into an array */
public void inputArray (Method inputMethod, MyInputStream stream)
{ try { // input number of elements and create an array of that size
    System.out.println("Enter number of elements");
    int n = stream.readInteger();
    a = new Object [n];
    // input the elements
    Object [] inputMethodArgs = {stream};
    for (int i = 0; i < n; i++)
        { System.out.println("Enter element " + (i+1));
          a[i] = inputMethod.invoke(null, inputMethodArgs);
        }
    } // end of try
catch (Exception e)
    { System.out.println(e);
      throw new IllegalArgumentException ("Array1D.inputArray");
    }
}
```



Table of Contents

- **Introduction**
- **Structure of a Java Program**
- **The Java Compiler and Virtual Machine**
- **Documentation Comments**
- **Data Types & Methods**
- **Exceptions**
- **Your Very Own Data Type**
- **Access Modifiers**
- **Inheritance and Method Overriding**
- **Defining an Exception Class**
- **Generic Methods**
- **Garbage Collection**
- **Recursion**
- **Testing and Debugging**



Garbage Collection

- Memory allocation
 - `int [] a = new int[100];`
`Currency c = new Currency();`
- Out of memory
 - Java's garbage collector is invoked
 - Checking the references in program variables
 - If not referenced any more
 - Garbage collection and memory reallocation
- Explicit garbage collection
 - Set the references to null
 - `a = null`
 - `b = null`



Table of Contents

- **Introduction**
- **Structure of a Java Program**
- **The Java Compiler and Virtual Machine**
- **Documentation Comments**
- **Data Types & Methods**
- **Exceptions**
- **Your Very Own Data Type**
- **Access Modifiers**
- **Inheritance and Method Overriding**
- **Defining an Exception Class**
- **Generic Methods**
- **Garbage Collection**
- **Recursion**
- **Testing and Debugging**



Recursive Functions

- Define a function in terms of itself

$$f(n) = \begin{cases} 1 & n \leq 1 \\ nf(n-1) & n > 1 \end{cases}$$

- Requirements
 - The definition must include a base component
 - Recursive component on the right side should have a parameter smaller than n
- $f(5) = 5 \times f(4) = 20 \times f(3) = 60 \times f(2) = 120 \times f(1)$



Induction (1/2)

- Proof
 - Induction base
 - Induction hypothesis
 - Induction step
- Example

$$\sum_{i=0}^n i = \frac{n(n+1)}{2}, n \geq 0$$



Induction (2/2)

- Induction base

$$n = 0, \sum_{i=0}^n i = 0$$

- Induction hypothesis

- We assume that equation is valid for $n \leq m$

- Induction step

$$n = m + 1, \sum_{i=0}^{m+1} i = m + 1 + \sum_{i=0}^m i = m + 1 + \frac{m(m+1)}{2} = \frac{(m+1)(m+2)}{2}$$

Recursive Methods for n!

- Java allows us to write recursive methods

```
public static int factorial(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return n * factorial(n - 1);  
}
```

```
factorial(5)  
= 5*factorial(4)  
= 5*4*factorial(3)  
... ..  
=5 * 4 * 3 * 2 * 1
```

Recursive Methods for sum

/ Generic sum method.**

*** @return null if array has no objects and sum of the objects a[0:n-1] otherwise */**

```
public static Computable sum (Computable [] a, int n)
{ if (a.length == 0) return null;
  Computable sum = (Computable) a[0].zero();
  for (int i = 0; i < n; i++) sum.increment(a[i]);
  return sum; }
```

/ Recursive Generic sum method.**

```
public static Computable recursiveSum (Computable [] a, int n) {
  if (a.length > 0) return rSum(a, n);
  else return null; // no elements to sum }
```

```
private static Computable rSum (Computable [] a, int n)
{ if (n == 0) return (Computable) a[0].zero();
  else return (Computable) rSum (a, n - 1).add(a[n-1]); }
```

Example 1.3

Recursive Method for permutations

```
/** perm(x, 0, n) outputs all permutations of x[0:n] */
public static void perm (Object [] list, int k, int m)
{ // Generate all permutations of list[k:m].
  int i;
  if (k == m) { // list[k:m] has one permutation, output it
    for (i = 0; i <= m; i++) System.out.print(list[i]);
    System.out.println();
  }
  else // list[k:m] has more than one permutation
    // generate these recursively
    for (i = k; i <= m; i++)
      { MyMath.swap(list, k, i);
        perm(list, k+1, m);
        MyMath.swap(list, k, i);
      }
}
```



Table of Contents

- **Introduction**
- **Structure of a Java Program**
- **The Java Compiler and Virtual Machine**
- **Documentation Comments**
- **Data Types & Methods**
- **Exceptions**
- **Your Very Own Data Type**
- **Access Modifiers**
- **Inheritance and Method Overriding**
- **Defining an Exception Class**
- **Generic Methods**
- **Garbage Collection**
- **Recursion**
- **Testing and Debugging**



What is Testing?

- Mathematical proof of correctness
 - Even a small program → quite difficult
- Program Testing
 - Test data
 - Subset of possible input data
 - Cannot cover all possible inputs
 - Objective of testing
 - To expose the presence of errors



Test Example: Quadratic Equation

```

public static void outputRoots(double a, double b, double c){
    if (a == 0) throw new IllegalArgumentException ("Coefficient of x^2 must be nonzero");
    double d = b * b - 4 * a * c;

    if (d > 0) {// two real roots
        double sqrtd = Math.sqrt(d);
        System.out.println ("2 real roots:" + (-b + sqrtd) / (2*a) + "and" + (-b - sqrtd) / (2*a));}

    else if (d == 0) // both roots are the same
        System.out.println ("1 distinct root: " + - b / (2 * a));

    else {// complex conjugate roots
        System.out.println("The roots are complex");
        System.out.println("The real part is " + - b / (2 * a));
        System.out.println("The imaginary part is " + Math.sqrt(-d) / (2 * a));
    }
}

```

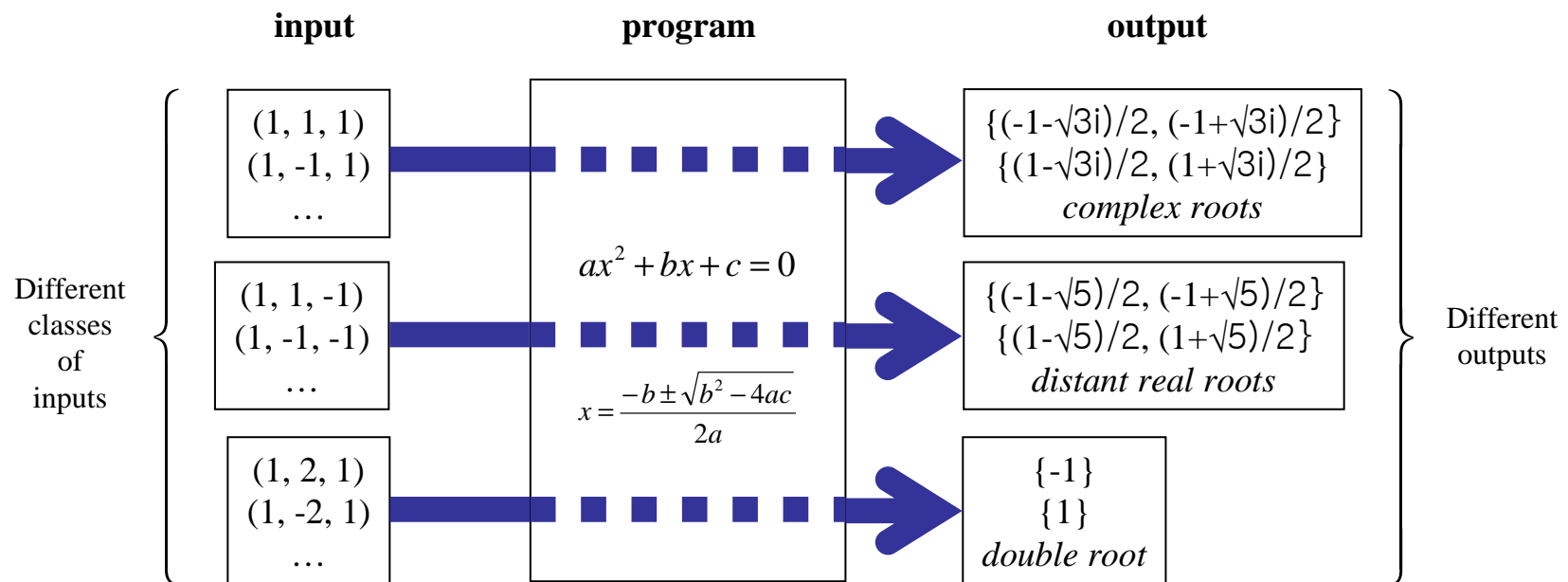


Designing Test data (1)

- Evaluating candidate test data
 - What is these data's potential to expose errors?
 - Can we verify the correctness of the program behavior on this data?
- Black box method
 - Partitioning data into qualitative different classes
 - Quadratic equation program
 - Test set → 3 classes
 - complex, real and distinct, real and the same roots
- White box method
 - Code based
 - Statement coverage
 - Every lines should be executed by test set
 - Decision coverage
 - Test set should cause each conditional in the program

Designing Test data (2)

- Black box method
 - Partitioning data into qualitative different classes



A test set should include at least one input from each class



Designing Test data (3)

- White Box Method
- Clause coverage
 - Strengthen the decision coverage
 - Boolean expression based
 - If ((C1 && C2) || (C3 && C4)) S1;
else S2
 - Make test set for C1, C2, C3, and C4 truth combination
- Execution coverage
 - Execution paths → order of statements executed in the program
 - Make test set for each execution path



Debugging

- Try to determine the cause of an error by logical reasoning
- Do not attempt to correct errors by creating special cases
- Be certain that your correction does not result in another error
- Begin with a single method that is independent of the others



Summary

- Requirements of program development
 - Representing data in an effective way
 - We need **efficient data structures**
 - Developing a suitable procedure
 - We need **good algorithms**
- Before you go further, you need to be
 - A proficient java programmer
 - An adept analyst of computer programs