Introduction to Java™

Module 1: Getting started, Java Basics

Prepared by Chris Panayiotou for EPL 602

What is Java?

Programming language developed by Sun Microsystems

- Simple
- Object Oriented
- Distributed
- Robust
- Secure
- Architecture Neutral
- Portable
- Interpreted
- High Performance
- Multithreaded
- Dynamic
- Network savvy

Why Java?

- The Java Language has many good design features – secure, safe (with respect to bugs), object-oriented, familiar (to C C++ and even Fortran programmers).
- Good set of libraries: networking, multimedia, from graphics to math functions.
- Best available electronic and paper training resources.
- Children will learn Java as it is a social language with natural graphical "hello world".

Why Java?

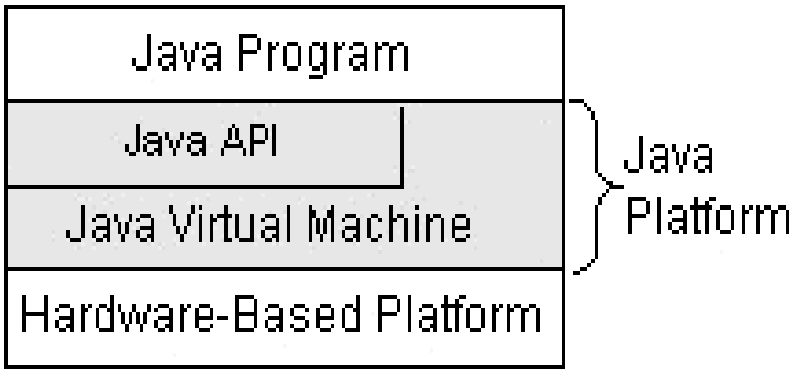
- Java is rapidly getting the better (the best!) Integrated Development Environments (IDEs) for programs.
- Java is naturally integrated with network and universal machine supports potentially powerful "write once-run anywhere" model.
- Easy to teach – I use it to help students understand client/server programming and concurrency – (dining philosophers).
- There is a large and growing trained labour force.

Java: Key Technical Ideas

- A Better Language:
  - Simplicity and C/C++ compatibility promote fluency;
  - GC and Threads allow software components;
  - Platform independence saves time;
  - Strong typing catches errors up front;
  - Declared exceptions forces coverage in code.
- Scalable Applications;
  - Threads for parallel speedup; patterns "in the large".
  - Dynamic linking allows simple apps to grow;
  - Range of implementations from JavaCard to HotSpot.

The Java platform

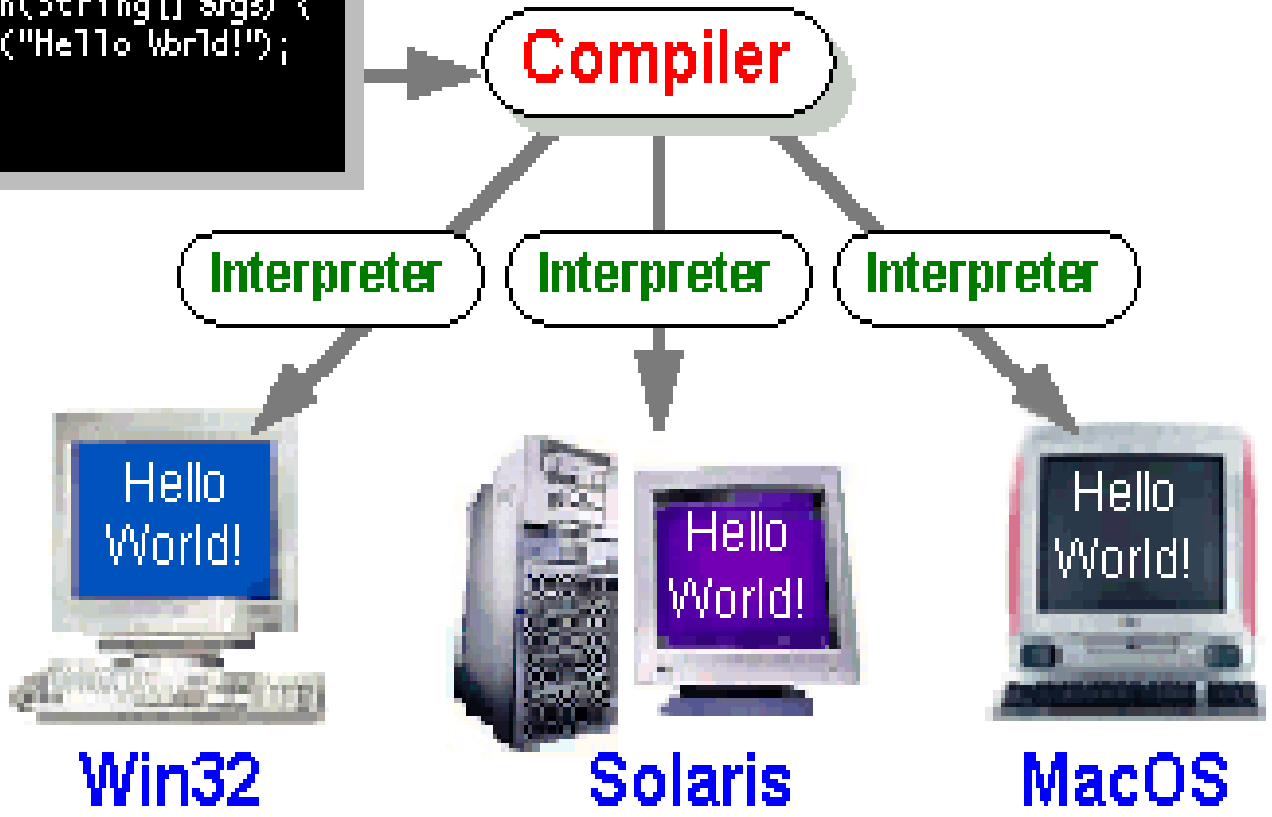
- The execution environment of Java programs
  - Java VM is platform dependent
  - Java API is platform independent

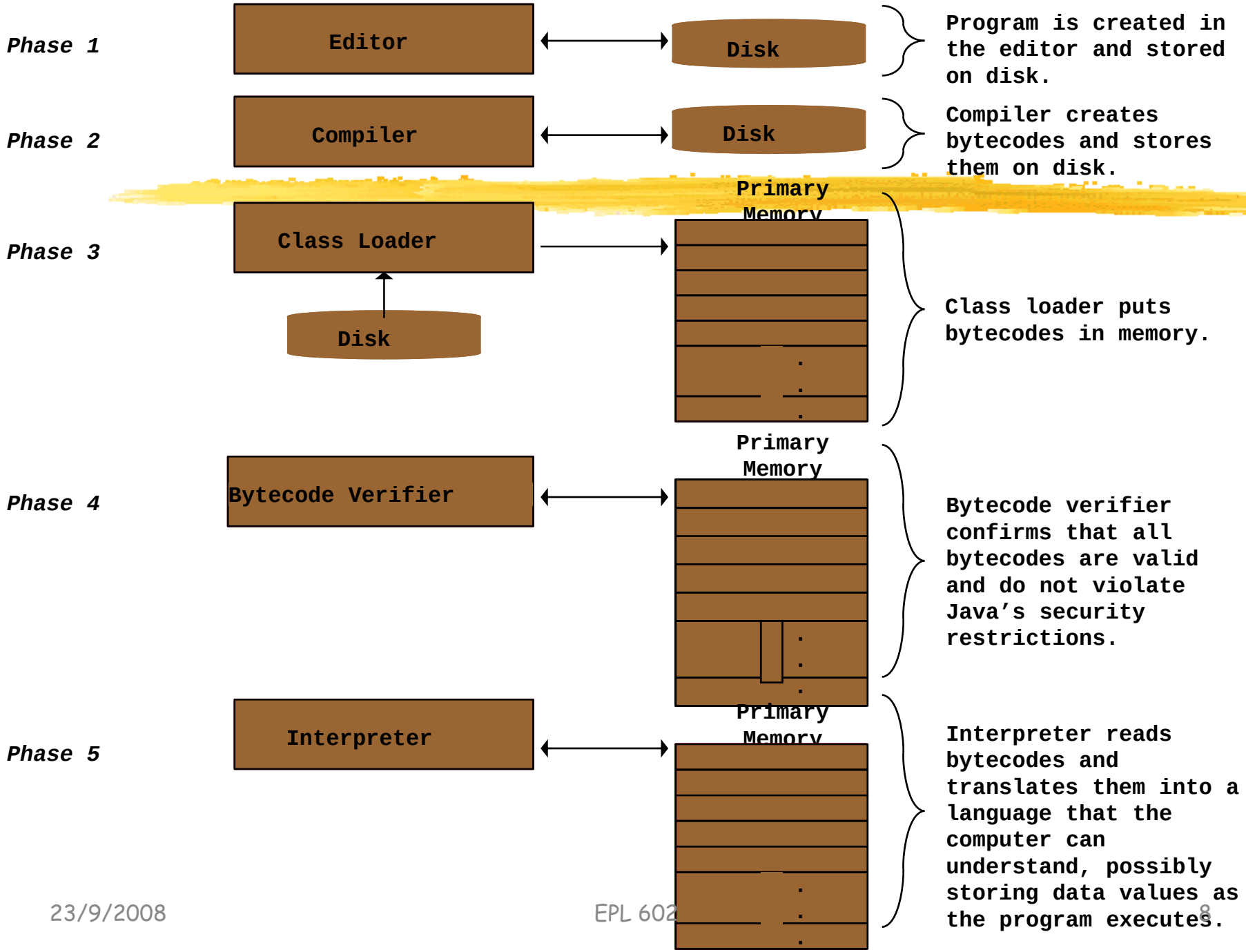


Java Program

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

HelloWorldApp.java




Phase 1
Editor
Disk
Program is created in the editor and stored on disk.
Phase 2
Compiler
Disk
Compiler creates bytecodes and stores them on disk.
Phase 3
Class Loader
Disk
Primary Memory
Class loader puts bytecodes in memory.
Phase 4
Bytecode Verifier
Primary Memory
Bytecode verifier confirms that all bytecodes are valid and do not violate Java's security restrictions.
Phase 5
Interpreter
Primary Memory
Interpreter reads bytecodes and translates them into a language that the computer can understand, possibly storing data values as the program executes.

Garbage collection

- Automatic Garbage Collection
  - Performed by the Java VM
  - Objects that are not referenced are removed
- To "destruct" an object manually make it equal to null. It will be collected by the Garbage Collector
- The Garbage Collector releases memory allocated with the new keyword (constructor call)

Classes and Packages

- Class
  - Defines an object
  - The same as in C++
- Package
  - Directory of classes
  - As libraries in C++

Java Documentation

- Can be downloaded from Sun and installed on the local computer
- Can be found online at http://java.sun.com/
- Contain the properties and methods of every object as well as the inheritance tree

Syntax & common commands

- Java syntax is very similar to that of C/C++
- Very common commands:
  - extends → inherits from (no multiple inheritance)
  - System.out.println(); → print to standard output
  - import → include
  - How import works:
    - ✓ **import java.awt.*; → will include all subclasses of the java.awt package with their methods**
    - ✓ **import java.awt.Button; → will include only the methods of the java.awt.Button class**

Java Syntax Tips

- Variables can be declared anywhere in the code.
  - They have to be initialized before use
  - Their scope is only within the code fragment enclosed by { } in which they were declared.
    Example: for(int i=0;i<10;i++)
    {← scope of i →}

- Foo x;
  x = new Foo();
  is equivalent to:
  Foo x = new Foo();

Program Syntax

- General Java Syntax:
- Class declaration:

```
public class myButton extends Button{
//variables, constructors and methods go here
}
```

public, private static,abstract, final class myButton extends Button implements interface

- Method declaration:

```
public void changeColor(Color x) {
//code goes here
}
```

public, private void, int, Integer, String … changeColor

Program Syntax

- Constructor declaration:
  ```
  public myButton() {
  //code to initialize the button
  }
  ```
  - No return type in the constructor.
  - The constructor is usually public, but not necessarily
- In a Java application (not applet) the main method must be declared. This is the first method executed when the program starts
- The syntax of the main method cannot be changed:
  `public static void main(String[] args) {…}`
  args is an array containing the command line arguments

Program Syntax

- General program syntax:

```
import java.awt.*;

public class myButton extends Button{
 int a;
String str;

public myButton() { //constructor
//construct the object
}

public void changeColor(){  //a method
 this.setColor(Color.red);
}
}//end class
```

Primitive types

- There are 8 primitive types in Java.
  - int vs Integer
  - float vs Float
  - double vs Double
  - long vs Long
  - short vs Short
  - char vs Character
  - boolean vs Boolean
  - byte vs Byte

Creating a Java program

- Required software:
  - Java Development Kit for your platform (includes the Java compiler, Java Virtual Machine, Appletviewer, Libraries and other utilities)
  - A text editor
- The **CLASSPATH** variable
  - Tells the Java compiler and Java virtual machine where the libraries are stored
  - It should include the current directory (.)
  - Usually set in the Autoexec.bat file

Creating a Java program

- Can contain directories and compressed files that contain classes or packages.
- Example:

  set CLASSPATH=c:\jdk1.1.2\lib\classes.zip;d:\myclasses\;.

- Adding the Java utilities in the PATH can ease your work
  - Example:

    set PATH=c:\dos;c:\windows;c:\jdk1.1.2\bin

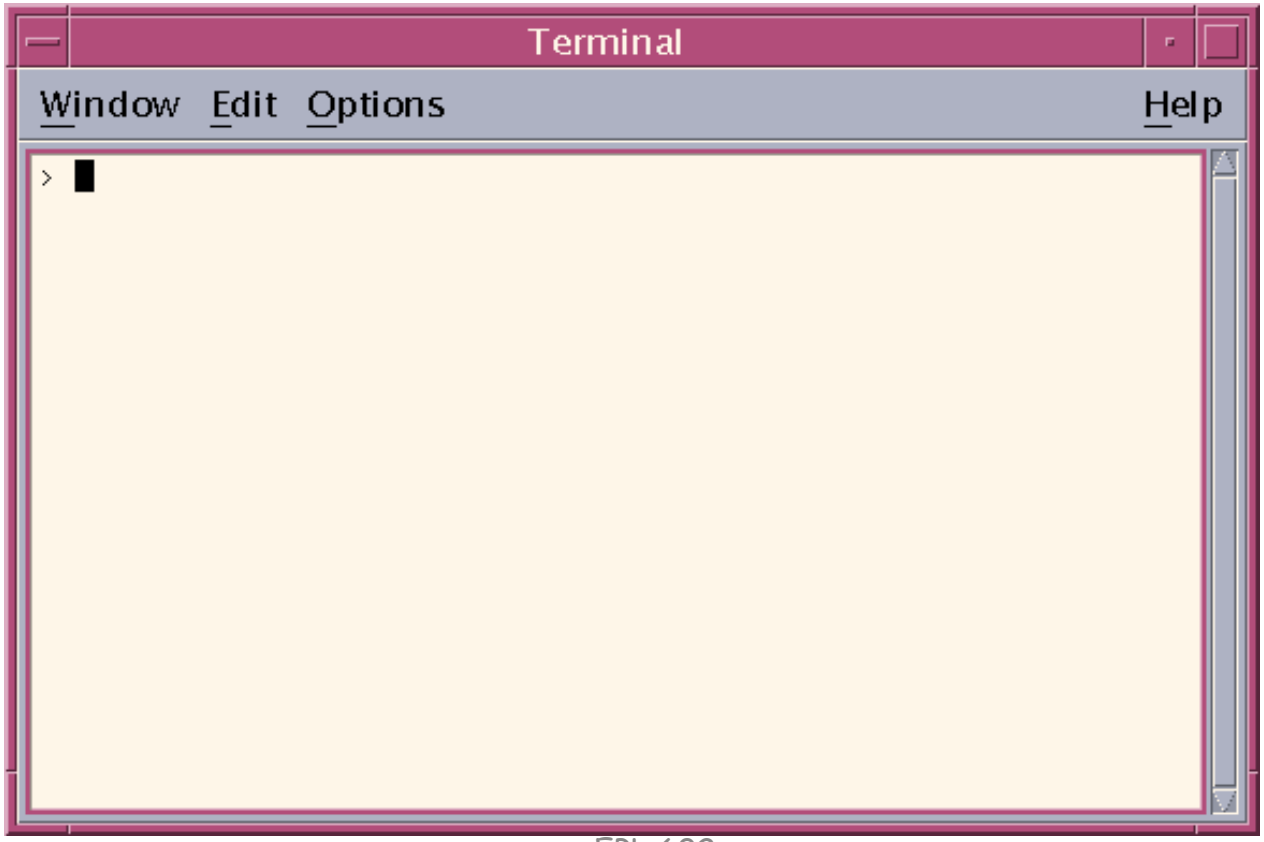
Creating Your First Application - HelloWorldApp

To create this program, you will:

- Create a Java source file (*.java).
- Compile the source file into a bytecode file. The Java *compiler*, javac, takes your source file and translates its text into instructions that the *Java Virtual Machine* (Java VM) can understand.
- Run the program contained in the bytecode file. The Java VM is implemented by a Java *interpreter*, java. This interpreter takes your bytecode file and carries out the instructions by translating them into instructions that your computer can understand.

Create a Java Source File.

- Bring up a *shell.*



Create a Java Source File

- The Java files you create should be kept in a separate directory (with mkdir).
- Start the editor.
- Type the code and store it in a file HelloWorldApp.java

HelloWorldApp.java

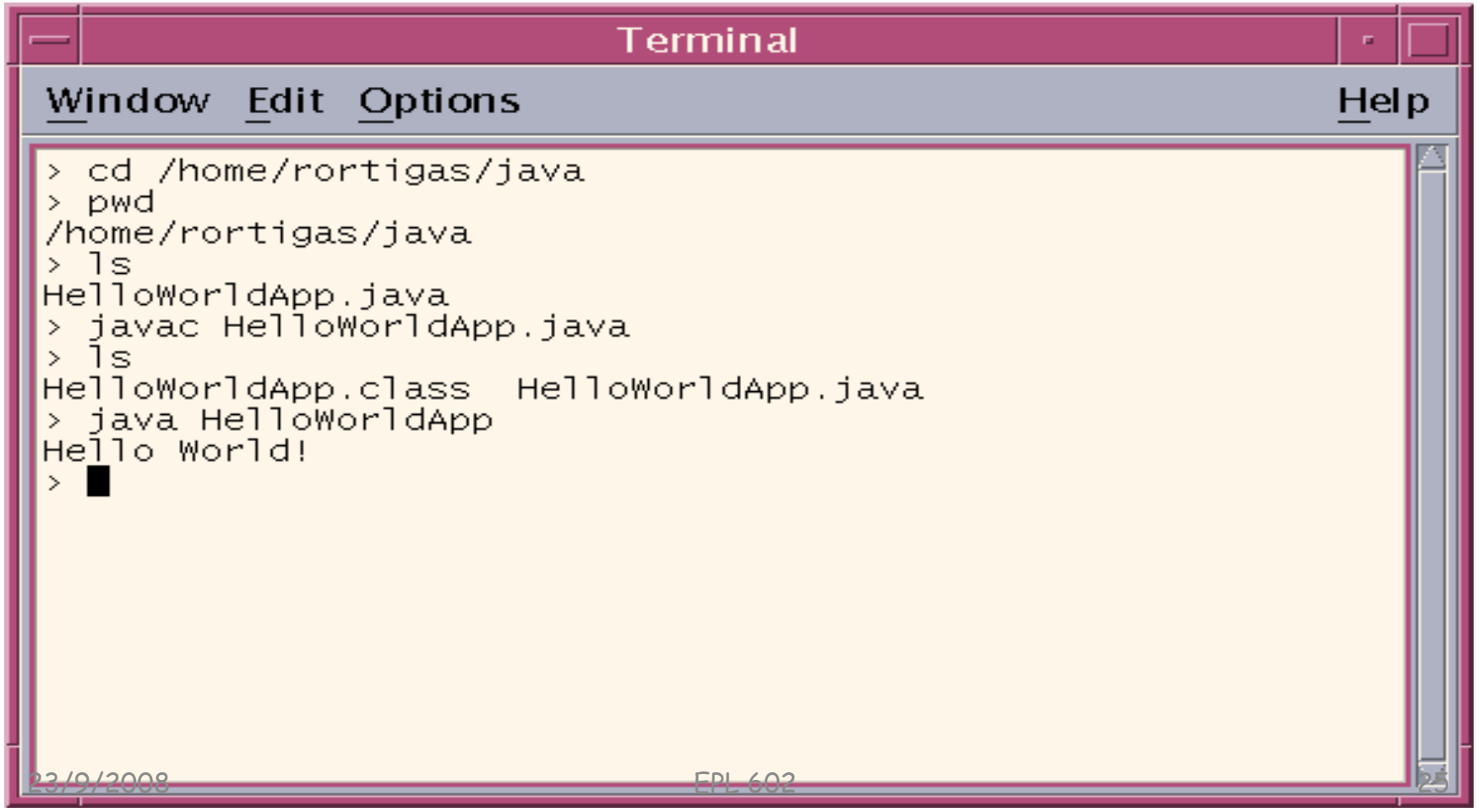
```
/**
  * The HelloWorldApp class implements
  * an application that simply displays
  * "Hello World!" to the standard output.
  */
 class HelloWorldApp {
  public static void main(String[] args)
  {
     System.out.println("Hello World!");
     }
 }
```

Compile the Source File

- javac *HelloWorldApp.java*
- If your prompt reappears without error messages, congratulations. You have successfully compiled your program.
- A *HelloWorldApp.class* file is created

Run the Program

- java HelloWorldApp



Useful classes

- String
- Hashtable
  Enables you to use objects other than numbers for indexing
- Vector
  A dynamically growing - shrinking array
- StringTokenizer
  For splitting a string into tokens (delimiter=space)
- StreamTokenizer
  For splitting a stream into tokens (delimiter can be chosen)

Introduction to Java™

Module 2: Expressions and Flow control

Prepared by Chris Panayiotou for EPL 602

Operators

- Used as in C/C++
  - + - * / =
  - % (modulo)
  - ! (not)
  - || (or), && (and)
  - <,>, ==, >=, <=, <>,!=
  - n++, n--, ++n, --n

Casting

- variable= (type) variable;
  - Example:

```
void casts() {
  int I = 200;
  long l = (long) I;
  long l2 = (long)200;
}
```

- Getting an int out of a String:

```
Integer x=new Integer(String);
int xValue= x.intValue();
```

Similar for float, double, long etc…
Look it up in the API documentation.

Casts in fundamental types

- Casting from another type to int, float, short, long, double:
  - Create an instance of the appropriate wrapper class using the suitable constructor
  - Get the desired value using the appropriate property, usually the .[type]Value
- Example: (String to float)

```
String piV="3.14159";
Float temp=new Float(piV); //create a new Float object
float pi = temp.floatValue(); //get the float value of the Float
```

Casting examples

- Integer

```
String n = "23";
Integer tmp = new Integer(n);
int i = tmp.intValue();
float f = tmp.floatValue();
double d = tmp.doubleValue();
Integer tmp2 = Integer.valueOf("342");
```

- Float

```
Float tmp = new Float(n);
int i = tmp.intValue();
float f = tmp.floatValue();
double d = tmp.doubleValue();
```

Flow control

- Java supports if-then-else, while, do-while, for and switch (case) statements as well as labeled breaks.
- The syntax for most of these statements is the same as in C/C++. (in addition variables can be declared in their conditions)
  - If-then-else

```
if (yoursales >=2*target)
 {bonus=1000;}
else if (yoursales >=1.5*target)
 {bonus=500;}
else if (yoursales >= target)
 {bonus=100;}
else {System.out.println("You are fired!!");}
```

Flow control

- while (just like C/C++)
  while (condiction) {block}

```
while(count<6)
 { System.out.println(count);
    count++;
 }
```

- do-while (just like C/C++)

```
do {block} while (condition)
```

Flow control

- for loops

```
for(statement1;condition;statement2)
 {statements;}
```

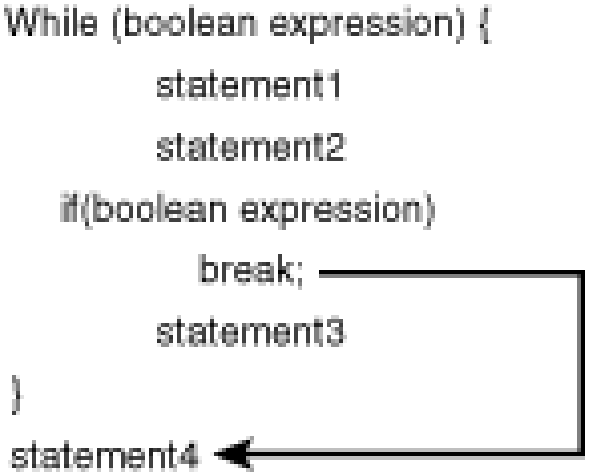
```
for(int x=0;x<100;x++)
  {System.out.println("Number is " + x);}
```

- switch statement

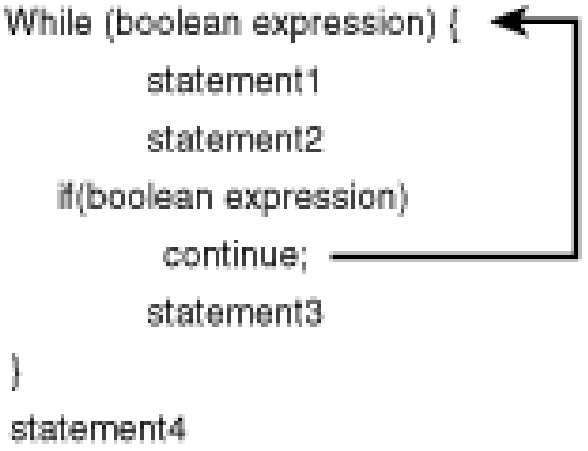
```
switch (choise) {
 case 1: {……;break;}
 case 2: {……;break;}
 default: {System.out.println("Invalid input");break;}
} //end switch
```

Flow control

- The break keyword
  - Used for breaking out of loops
    (and in the switch statement)



- The continue statement
  - Used for transferring the execution to the top of the loop



Flow control

- Labeled break/continue
  - Similar to GOTO
  - Convenient for breaking out of nested loops or transferring the execution at the top of the loop.
  - Label must be placed just outside the loop followed by :
  - Works with all kinds of loops

Flow control

- **break/continue example**

```
outsideLoop: //label
   for(int out=0; out<3; out++) {
      //some code
    for(int inner=0;inner < 5; inner++) {
      //some code
      if (.....)
       {break outsideLoop;} //break out of the outer loop
   if (.....)
    {continue outsideLoop;} //continue to outer loop
    }//inner loop
   }//outer loop
   System.out.println("All done");
```

Introduction to Java™

Module 3: Arrays

Prepared by Chris Panayiotou for EPL 602

An Introduction to Arrays

- Arrays in Java are just as they exist on all programming languages
  - Come in handy when you want to organize multiple values of logically related items
  - They are static objects (meaning: Cannot be resized dynamically)
- The syntax used for arrays is: identifier[subscript] (eg. Array1[5])
  - The identifier refers to the array as a whole
  - While the subscript refers to a specific element of the array

Creating Arrays

- To create an array first you must declare it :
  `int numbers[];` or `int[] numbers;`
- Java lets you create arrays only using the new operator, like this:
  `number= new int[x];`
  `// x is an integer stating the size of the array`
- For primitive types that is enough…
- However for an array of some other class type there is the need to initialize every object of your array manually
- This is necessary because so far you have an array of null objects

Initializing an Array

- To properly initialize an array (not of primitive type) you have to initialize every element of the array like this:

```
MyStrangeObject[] objs;
objs = new MyStrangeObject[20];
for (int I=0;I<20;I++)
    objs[I] = new MyStrangeObject();
```

- **NOTE**
  objs[**0**]; is the first element of the array.

More on Arrays

- **CAUTION**

  Don't try to access a nonexistent array element. For example:

```
int numbers[];
numbers = new int[10];

n = numbers[15];
```

  would go beyond the boundaries of the array and so Java would generate the exception ArrayIndexOutOfBounds

Number of Elements Vs Array Size

- The member length gives the size of an array.
- An array may have less elements than its size.
- To find the number of elements in an array a counter should be used.
- Alternatively use a for loop to check how many positions of the array are initialized.

```
for (int I=0;I<nums.length;I++)
    if (nums[I]!=null) counter++;
```

Copying an Array

- To copy an array one should
  - Create a new array with the size of the origin array:
    ```
    String[] copy=new String[origin.length()];
    ```
  - Copy each object of the origin array manually:

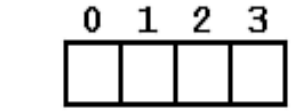
    ```
    for (int I=0;I<origin.length();I++)
          copy[I]=new String(origin[I]);
    ```
- The following code would result to reference to the same array:

```
String[] origin;
String[] copy;
origin = new {“one”, “two”, “three”};
copy = origin;
// A common mistake where one would think that
// he has two different arrays
```

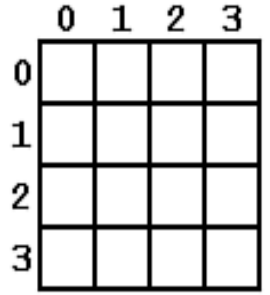
Multidimensional Arrays

- Java doesn't support multidimensional arrays in the conventional sense
- Possible to create arrays of arrays
- To create a two-dimensional array of integers you would write something like this:

```
int table[][] = new int[4][4];
```



One-Dimensional Array



Two-Dimensional Array

Multidimensional Arrays

- You refer to a value stored in a n-dimensional array by using subscripts for all the dimensions like this:
  `int value = table[3]…[2];`
- A quick way to initialize an n-dimensional array is to use nested for loops

A Bit About Collections

- Collections vs Arrays
  - Dynamic sizing
  - Any type of object can be put in a collection
  - No support for primitive types
- Vector: Like a dynamic array of objects
- BitSet: A vector of bits (minimum size 64 bits)
- Hashtable: An *associative* dynamic array (links a key object with a value object)
- Stack: A last-in, first-out (LIFO) collection with push and pop methods

Introduction to Java™

Module 4: Objects and Classes

Prepared by Chris Panayiotou for EPL 602

Encapsulation, Polymorphism and Inheritance

- Classic OOP concepts
- Encapsulation: Binding together the properties of an item → creating an object combing of objects
- Polymorphism: Dynamic binding of types
  - An object B which is a subclass of object A can be handled by a type A member field
  - Thus we can call method M of class A from object B
- Inheritance: Refining a base class
  - A new class is derived from the base class
  - The accessible methods and fields of the base class are inherited to the new class
  - Inherited methods can be overridden

Using Objects

- Creating an object
  - Write its class – probably by subclassing
  - Write one (or more) constructor(s)
  - Write its methods
  - Example object:

```
class myObject extends Object {
int a;                    // a member field

myObject() {…}            // default constructor
myObject(int i) {…}       // another constructor

void mehtod1() {…}        // a method that does something

…
}
```

Using Objects

- Initializing an object
  - Create it using the new operator
  - During the creation a constructor is called

```
anObject = new myObject();
```

- Calling an object's methods

```
anObject.method1();
// Calls method1 of the anObject object
```

Defining Methods

- Methods are like functions are in C
- To define a method first we declare what object type it returns (e.g. String toString() {..})
- We can use any of the access modifiers to limit the access to that method (e.g. public String toString() {..})
- We can declare its parameters if any (e.g. public String toString(String s) {..})
- Finally we can use any other modifiers we wish: final, abstract, synchronized, static
  (e.g. public static String toString() {..})
  (e.g. public final String toString(String s) {..})

The “this” Keyword

- The “this” keyword can be used to access the methods and member fields of the current object.
- It is a handle of the current object.

```
public class foo {
    int aInt = 2;

    foo() {…}
    public int aMehtod() {…}
    public foo getMe() { return this; }
    public static void main(String[] args){
          this.aInt = this.aMethod();
    }
}
```

Access Modifiers

- "Friendly"
  - No modifier is used - default access limitations
  - Only classes in the same package can use it
  - Cannot be inherited by subclasses in foreign packages
- Public
  - Allows access by everyone
- Private
  - Access is forbidden to everybody except the owner class
- Protected
  - Access limitations are the same as in the "friendly" case
  - Can be inherited by subclasses in foreign packages

Applicable in: classes*, constructors, methods, fields

Subclassing

- Subclassing involves two classes: the base class and the newly created derived class.
- When subclassing we inherit from the base class into the derived class
- Creating a subclass:

```
public class foo extends goo {
int I;

   foo () {
     super();
     I=0;
   }
} // This is a subclass of class goo
```

Overriding methods

- **Methods declared in the subclass as well as the superclass.**
- **When called, the method in the subclass (not the superclass) will be executed.**

Example:

```
class One {
            //constructor
               public void method1() {
                System.out.println("This is class One"); }
               }
class Two extends One {
                                  …….
Overriding method →    public void method1() {
                                  System.out.println("This is class Two");
}
```

Interfaces

- The **interface** keyword takes the **abstract** concept one step further ("pure" **abstract** class)
- An **interface** provides only a form, **but no implementation.**
  - It allows the creator to establish the form for a class: method names, argument lists, and return types, but no method bodies.
- **Interfaces** can contain fields that are implicitly **static** and **final**.
  - Automatically **static** and **final** → cannot be "blank finals"
  - Can be initialized with nonconstant expressions.

Inheritance and Interfaces

- As an **interface** has no implementation at all many **interfaces** can being combined (implemented) to form a new derived class
- Valuable when you need to say "An **x** is an **a** *and* a **b** *and* a **c**."
  - In C++ → *multiple inheritance*
    - ✓ Carries some rather sticky baggage because each class can have an implementation.
  - In Java → can perform the same act, but only one of the classes can have an implementation.
    - ✓ So the problems seen in C++ do not occur with Java when combining multiple interfaces

Interface example

```
// Multiple interfaces.
import java.util.*;
interface CanFight { void fight(); }
interface CanSwim { void swim(); }
interface CanFly { void fly(); }

class ActionCharacter { public void fight() {} }
class Hero extends ActionCharacter implements CanFight, CanSwim, CanFly {
   public void swim() {}
   public void fly() {} }
public class Adventure {
   static void t(CanFight x) { x.fight(); }
   static void u(CanSwim x) { x.swim(); }
   static void v(CanFly x) { x.fly(); }
   static void w(ActionCharacter x) { x.fight(); }
   public static void main(String[] args) {
        Hero h = new Hero();
        t(h); // Treat it as a CanFight
        u(h); // Treat it as a CanSwim
        v(h); // Treat it as a CanFly
        w(h); // Treat it as an ActionCharacter
   }
}
```

Introduction to Java™

Module 5: Advanced Language Features

Prepared by Chris Panayiotou for EPL 602

Constructors

- **Constructor is the method called when an object is initialized with the new keyword**
- **Constructors can also be overridden (they usually are)**
- **Constructors can invoke constructors from the superclass**

Example:

```
public class redCircle extends Circle {……
        //constructor
        public redCircle(int x, int y,int radius) {
                super(x,y,radius);
                color=Color.red;
        }
}
```

Static variables & methods

- Static methods
  - No need to create an instance of the class containing the method to use it
    Example:
    double x = Console.readDouble();
    double x = Math.pow(3, 0.1);
    This works because the method readDouble of the Console class and the method pow of the Math class are declared as static

    I.e. public static double readDouble() {……}

Static variable & methods

- Static variables
  - No need to create an instance of the class to access them

Example:

```
public class defaults {
   public static String hostname="java.sun.com"
……}

public class anotherClass {
  ……
  System.out.println("The hostname is " +
defaults.hostname);
  ……}
```

Final classes, variables and methods

- A final class cannot become a parent class
  Example: final class Card{ …… }
- Specific methods in a class can be declared as final. A final method cannot be overridden
  Example: public final void doSomethind() {……}
- A final variable cannot change value
  Example: public final int x=15;
- Using final improves performance

Abstract classes & methods

- An abstract method is basically only a method's declaration. (something like the .h files in C/C++)
  Example: public abstract void play();
- Declaring a method as abstract, is promising that all non-abstract descendants of the abstract class will implement that abstract method.
- An abstract class contains at least one abstract method
  Example: public abstract class Message {.....}

Wrapper classes

- An instance of the Double class wraps the double type, Integer the int type and so on…

  Example:

  Suppose we need a vector of Double. Simply adding numbers to the vector won't work:

```
Vector v = new Vector();
v.addElement(3.14); //ERROR
```

  The floating-point number 3.14 is not an object. Here we can use the Double wrapper class to create a Double object and add it to the vector: `v.addElement(new Double(3.14));`

Working with Strings

- Java.lang.String
- Create:
  String x = new String("a string");
- Concat:
  x=x+", another string"; (x="a string, another string")
- Length:
  int stringLength = x.length();
- Comparing
  if (x.equals("a string"))
  if (x.compareTo("a string"))

Working with Strings

- Other useful methods:
  - indexOf(String)
    - ✓ Returns the index within this string of the first occurrence of the specified character.
  - replace(char, char)
    - ✓ Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar
  - startsWith(String)
    - ✓ Tests if this string starts with the specified prefix.
  - trim()
    - ✓ Removes white space from both ends of this string.
- Check the API documentation for the complete list of functions

Working with Strings

- Java provides two classes for working with Strings
  - *String* and *StringBuffer*
- Class String is used for strings that remain constant (their value doesn't change)
- Class StringBuffer is used for strings that may change
  - E.g. Reading the contents of a text file
  - Check out the Java API for StringBuffer
- Using String is more efficient when our strings remain unchanged as they are constants that can be jointly used by other code in our program
- StringBuffer is more efficient when we have changing strings as we create only one object

Working with Strings

```
class SameString {
    public static void main(String[] args) {
        String s1 = "dog";
        String s2 = "It's a dog's life";
        String s3 = "dog";
        if (s1 == s2) System.out.println("s1 == s2"); // FALSE
        if (s1 == s3) System.out.println("s1 == s3"); // TRUE
        if (s1 == "dog")  System.out.println("s1 == \" dog \""); //TRUE
        String doggy = new String(s1);
        if (s1 == doggy) System.out.println("s1 == doggy"); //FALSE
    }
}
```

Introduction to Java™

Module 6: Exceptions

Prepared by Chris Panayiotou for EPL 602

Error Handling

- The basic philosophy of Java is that **"badly-formed code will not be run."**
- As with C++, the ideal time to catch the error is at compile time.
- However, not all errors can be detected at compile time.
- The rest of the problems must be handled at run-time.
- Some formality allows the originator of the error to pass appropriate information to a recipient who will know how to handle the difficulty properly.

Error Handling

- In C and other earlier languages there could be several of these formalities.
- They were generally established by convention and not as part of the programming language.
- Typically, you returned a special value or set a flag, and the recipient was supposed to look at the value or the flag.
- However, programmers who use a library tend to think of themselves as invincible: "Yes, errors might happen to others but not in ***my*** code."
- This approach to handling errors was a major limitation to creating large, robust, maintainable programs.

Error Handling

```
errorCodeType readFile {
        initialize errorCode = 0;
        open the file;
        if (theFileIsOpen) {
            determine the length of the file;
            if (gotTheFileLength) {
                allocate that much memory;
                if (gotEnoughMemory) {
                    read the file into memory;
                    if (readFailed) {
                        errorCode = -1;
                    }
                } else {errorCode = -2;}
            } else {errorCode = -3; }
            close the file;
            if (theFileDidntClose && errorCode == 0) {
                errorCode = -4;
            } else {
                errorCode = errorCode and -4;
            }
        } else {errorCode = -5;}
        return errorCode;
    }
```

```
readFile {
        try {
            open the file;
            determine its size;
            allocate that much memory;
            read the file into memory;
            close the file;
        } catch (fileOpenFailed) {
            doSomething;
        } catch (sizeDeterminationFailed) {
            doSomething;
        } catch (memoryAllocationFailed) {
            doSomething;
        } catch (readFailed) {
            doSomething;
        } catch (fileCloseFailed) {
            doSomething;
        }
    }
```

Error Handling with Exceptions

- Exception handling is enforced by the Java compiler.
- You can generate your own exceptions.
- An *exceptional condition* is a problem that prevents the continuation of the method or scope that you're in.
  - With an exceptional condition, you cannot continue processing.
  - All you can do is jump out of the current context and relegate that problem to a higher context.
  - This is what happens when you throw an exception.

Error Handling with Exceptions

- At the point where the problem occurs:
  - You might not know what to do with it
  - You must stop and somebody, somewhere, must figure out what to do.
  - Don't have enough information in the current context to fix the problem.
  - You hand the problem out to a higher context where someone is qualified to make the proper decision (much like a chain of command).

Exceptions. So What?

- A significant benefit of exceptions is that they clean up error handling code.
- No need of checking for a particular error and dealing with it at multiple places in your program.
- The exception will guarantee that someone catches it.
- Need to handle the problem in only one place, the so-called *exception handler*.
- This saves you code and it separates the code that describes what you want to do from the code that is executed when things go awry.
- In general, reading, writing, and debugging code becomes much clearer with exceptions.

Exception Specification

- In Java you have to inform of the exceptions that might be thrown from your method.
  - Java provides syntax (and *forces* you to use that syntax) for that
  - This is the *exception specification* and it's part of the method declaration. So your method definition might look like this:

```
void f() throws tooBig, tooSmall {
  // Some code
}
```

- exceptions of type **RuntimeException** can reasonably be thrown anywhere
- Java guarantees that exception correctness can be ensured *at compile time*.

Throwing an Exception

- Throw a different class of exception for each different type of error.
- When you throw an exception, several things happen:
  - The exception object is created in the same way that any Java object is created: on the heap, with **new**.
  - The current path of execution is stopped and the handle for the exception object is ejected from the current context.
  - The exception-handling mechanism takes over and begins to look for an appropriate place to continue executing the program.
  - This appropriate place is the *exception handler.*
  - The *exception handler* recovers from the problem so the program can either try another task or simply continue.

Throwing an Exception

- Example of throwing an exception:
  - Consider an object handle called t.
  - You might want to check before trying to call a method using that object handle if it is valid.
  - You can send information about the error into a larger context by creating an object representing your information and "throwing" it out of your current context.
  - This is called *throwing an exception*. Here's what it looks like:

```
if(t == null)
  throw new NullPointerException();
```

Catching an Exception

- If a method throws an exception, it must assume that exception is caught and dealt with.
- The try block
  - If inside a method an exception is thrown that method will exit in the process of throwing.
  - To avoid this you can set up a special block within that method to capture the exception.
  - This is called the *try block* because you "try" your various method calls there. Example:

```
try {
  // Code that might generate exceptions
}
```

Catching an Exception

- Thrown exceptions go to an *exception handler*
- Multiple *exception handlers* - one for every exception type you want to catch
- Exception handlers immediately follow the try block and are denoted by the keyword **catch**:

```
try {
    // Code that might generate exceptions
} catch(Type1 id1) {
    // Handle exceptions of Type1
} catch(Type2 id2) {
    // Handle exceptions of Type2
}
// etc...
```

Catching any Exception

- It is possible to create a handler that catches any type of exception.
- You do this by catching the base-class exception type **Exception**:

```
catch(Exception e) {
  System.out.println("caught an exception");
}
```

- The Exception class has the following methods:
  - String getMessage( )
  - String toString( )
  - void printStackTrace( )
    void printStackTrace(PrintStream)

Performing Cleanup with Finally

- If you want to execute some code whether or not an exception occurs in a try block you use a finally clause at the end of all the exception handlers:

```
try {
 // Dangerous stuff that might throw A or B
} catch (A a1) {
  // Handle A
} catch (B b1) {
  // Handle B
} finally {
  // Stuff that happens every time
}
```

- Whether an exception is thrown or not, the finally clause is always executed.

What's finally for?

- **finally** is necessary when you need to set something *other* than memory back to its original state.
- This is usually something like an open file or network connection, something you've drawn on the screen etc.
- Even in cases in which the exception is not caught in the current set of **catch** clauses, **finally** will be executed before the exception-handling mechanism continues its search for a handler at the next higher level.
- The **finally** statement will also be executed in situations in which **break** and **continue** statements are involved.

Failing to Catch an Exception

- Possible only for **RuntimeExceptions**.
- If a RuntimeException gets all the way out to **main( )** without being caught, printStackTrace( ) is called for that exception as the program exits.
- Keep in mind that it's possible to ignore only **RuntimeExceptions**, since all other handling is carefully enforced by the compiler.
- A **RuntimeException** represents a programming error:
  - An error you cannot catch (e.g. receiving a null handle handed to your method by a client programmer)
  - An error that you should have checked for in your code (such as ArrayIndexOutOfBoundsException where you should have paid attention to the size of the array).

Re-throwing an Exception

- You can re-throw the exception that you just caught:

```
catch(Exception e) {  System.out.println("An
exception was thrown");
  throw e;
}
```

  - Re-throwing an exception causes the exception to go to the exception handlers in the next-higher context.
  - Any further catch clauses for the same try block are still ignored.
  - Possible to re-throw a different exception from the one you caught.

Creating Your Own Exceptions

- Need to create your own exceptions to denote a special error that your library is capable of creating.
- To create your own exception class, you're forced to inherit from an existing type of exception.
- Inheriting an exception is quite simple:

```
class MyException extends Exception {
  public MyException() {}
  public MyException(String msg) { super(msg);}
}
```

Exception Restrictions

- When you override a method, you can throw only the exceptions that have been specified in the base-class version of the method.
- This is a useful restriction, since it means that code that works with the base class will automatically work with any object derived from the base class (a fundamental OOP concept, of course), including exceptions.