CMSC 341

Java Packages, Classes, Variables, Expressions, Flow Control, and Exceptions

Sun's Naming Conventions

- Classes and Interfaces

  `StringBuffer, Integer, MyDate`

- Identifiers for methods, fields, and variables

  ```
  _name, getName, setName, isName,
    birthDate
  ```

- Packages

  `java.lang, java.util, proj1`

- Constants

  `PI, MAX_NUMBER`

Comments

- Java supports three types of comments.
  - C style   /* multi-liner comments */
  - C++ style   // one liner comments
  - Javadoc

```
/**
    This is an example of a javadoc comment.  These
    comments can be converted to part of the pages you
    see in the API.
*/
```

The `final` modifier

- Constants in Java are created using the *final* modifier.

```
final int MAX = 9;
```

- Final may also be applied to methods in which case it means the method can not be overridden in subclasses.
- Final may also be applied to classes in which case it means the class can not be extended or subclassed as in the String class.

Packages

- Only one package per file.
- Packages serve as a namespace in Java and create a directory hierarchy when compiled.
- Classes are placed in a package using the following syntax in the first line that is not a comment.

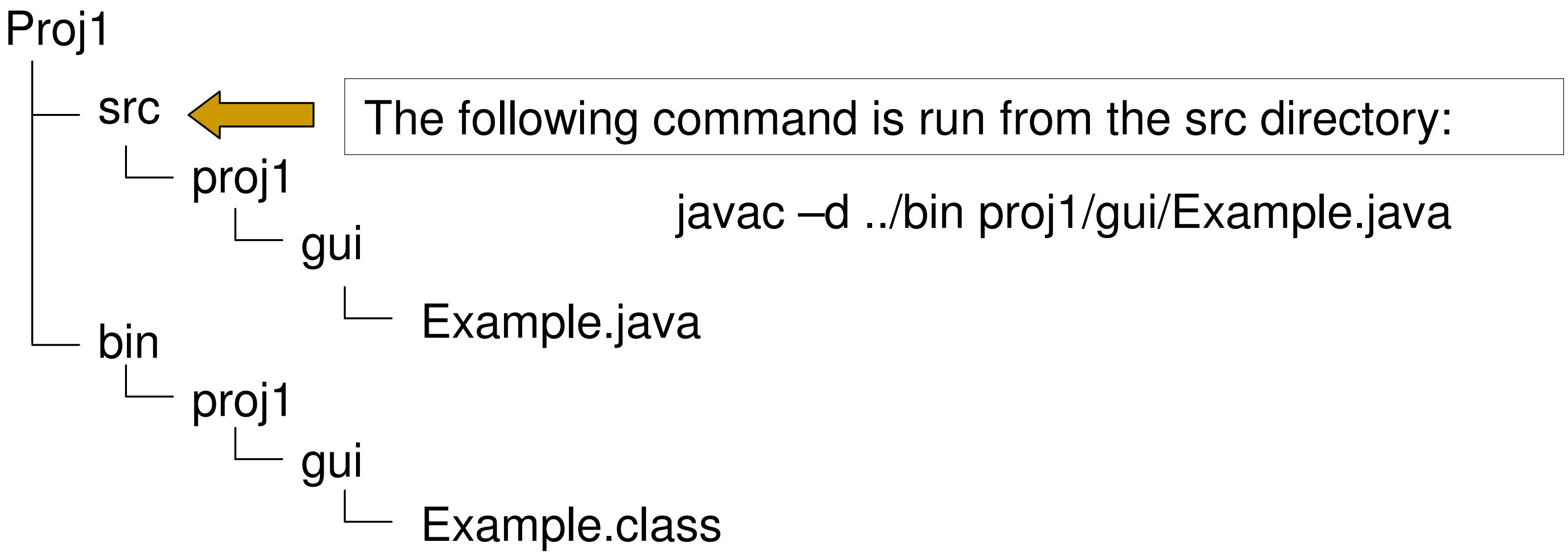
```
package packagename;

package packagename.subpackagename;
```

Packages (cont.)

- Classes in a package are compiled using the –*d* option.
- On the following slide, you will find the command to compile the code from the `Proj1/src` directory to the `Proj1/bin` directory.

Packages (cont.)

- It is common practice to duplicate the package directory hierarchy in a directory named *src* and to compile to a directory named *bin*.



```
Proj1
 ├─ src
 │   └─ proj1
 │       └─ gui
 │           └─ Example.java
 └─ bin
     └─ proj1
         └─ gui
             └─ Example.class
```

The following command is run from the src directory:

```
javac –d ../bin proj1/gui/Example.java
```

Packages (cont.)

- By default, all classes that do not contain a package declaration are in the unnamed package.
- The fully qualified name of a class is the *packageName.ClassName*.

```
  java.lang.String
```

- To alleviate the burden of using the fully qualified name of a class, people use an import statement found before the class declaration.

```
  import java.util.StringBuffer;
  import java.util.*;
```

Fields and Methods

- In Java you have fields and methods. A field is like a data member in C++.
- Method is like a member method in C++.
- Every field and method has an access level. The public, private, and protected keywords have the same functionality as those in C++.
  - `public`
  - `protected`
  - `private`
  - (package)

Access Control

Modifier	*Same class*	*Same package*	*Subclass*	*Universe*
`private`	✓			
default	✓	✓		
`protected`	✓	✓	✓	
`public`	✓	✓	✓	✓

Access Control for Classes

- Classes may have either public or package accessibility.
- Only one public class per file.
- Omitting the access modifier prior to class keyword gives the class package accessibility.

Classes

- In Java, all classes at some point in their inheritance hierarchy are subclasses of java.lang.Object, therefore all objects have some inherited, default implementation before you begin to code them.
  - `String toString()`
  - `boolean equals(Object o)`

Classes (cont.)

- Unlike C++ you must define the accessibility for every field and every method. In the following code, the x is public but the y gets the default accessibility of package since it doesn't have a modifier.

```
public
  int x;
  int y;
```

Instance and Local Variables

- Unlike C++ you must define everything within a class.
- Like C++,
  - variables declared outside of method are instance variables and store instance or object data. The lifetime of the variable is the lifetime of the instance.
  - variables declared within a method, including the parameter variables, are local variables. The lifetime of the variable is the lifetime of the method.

Static Variables

- A class may also contain static variables and methods.
- Similar to C++…
    - Static variables store static or class data, meaning only one copy of the data is shared by all objects of the class.
    - Static methods do not have access to instance variables, but they do have access to static variables.
    - Instance methods also have access to static variables.

Instance vs. Static Methods

- Static methods
  - have *static* as a modifier,
  - can access static data,
  - can be invoked by a host object or simply by using the class name as a qualifier.
- Instance methods
  - can access static data,
  - can access instance data of the host object,
  - must be invoked by a host object,
  - contain a `this` reference that stores the address of host object.

Pass By Value or By Reference?

- All arguments are passed by value to a method. However, since references are addresses, in reality, they are passed by reference, meaning…
  - Arguments that contain primitive data are passed by value. Changes to parameters in method do not effect arguments.
  - Arguments that contain reference data are passed by reference. Changes to parameter in method may effect arguments.

Constructors

- Similar to C++, Java will provide a default (no argument) constructor if one is not defined in the class.
- Java, however, will initialize all fields (object or instance data) to their zero values as in the array objects.
- Like C++, once any constructor is defined, the default constructor is lost unless explicitly defined in the class.

Constructors (cont.)

- Similar to C++, constructors in Java
  - have no return value,
  - have the same name as the class,
  - initialize the data,
  - and are typically overloaded.
- Unlike C++, a Java constructor can call another constructor using a call to a `this` method as the first line of code in the constructor.

Expressions and Control Flow

- Java uses the same operators as C++. Only differences are
  - + sign can be used for String concatenation,
  - logical and relative operators return a `boolean`.
- Same control flow constructs as C++, but expression must return a `boolean`.
  - Conditional
    - `if`(<boolean expression>){…}`else if`(<boolean expression>){…}`else`{…}
    - `switch`(variable){`case` 1: …`break`; `default`:…}
      - Variable must be an integral primitive type of size int or smaller, or a char

Control Flow Constructs (cont.)

- Iterative
  - while (<boolean expression>) { … }
  - do { … } while (<boolean expression>);
  - for(<initialize>; <boolean expression>; <update>) { … }
  - break and continue work in the same way as in C++.
  - May use labels with break and continue as in C++.

Control Flow Constructs (cont.)

- Enhanced for loop since Java 5 for iterating over arrays and collections.

```
public class EnhancedLoop
{
   public static void main(String []a )
   {
     Integer [] array = {new Integer(5),6,7,8,9};
     for (int element: array){
            element+= 10;
            System.out.println(element);
     }
     for (int element: array){
            System.out.println(element);
     }
   }
}
```

← element is a local variable

Example Class

```
public class Person
{
   // instance data
   private String name;
   private int age = 21;
   private static int drivingAge = 16;
   private static int num = 0;

   //constructors
   public Person(String name)
   {
        this.name = name;
        num++;
   }
   public Person(String name, int age){
        this(name);
        this.age = age;
   }
```

C++ style comments

static `num` tracks the number of Person objects

Call to previous constructor

Example Class (cont.)

```
//accessor and mutators
public String getName(){
    return name;
}
public void setName(int name){
    this.name = name;
}
public int getAge(){
    return age;
}
public void setAge(int age){
    this.age = age;
}
```

The `this` reference is used to differentiate between local and instance data

Example Class (cont.)

C style comments

```
/* static accessor methods
   The this reference does not
   exist in static methods
*/
public static int getDrivingAge(){
   return drivingAge;
}
public static int getNum(){
   return num;
}
```

Example Class (cont.)

```
  //overridden methods inherited from Object
  public String toString(){
      return "Person " + name;
  }
  public boolean equals(Object o){
      if( o == null)
          return false;
      if( getClass() != o.getClass())
          return false;
      Person p = (Person)o;
      return this.age == p.age;
  }
}
```

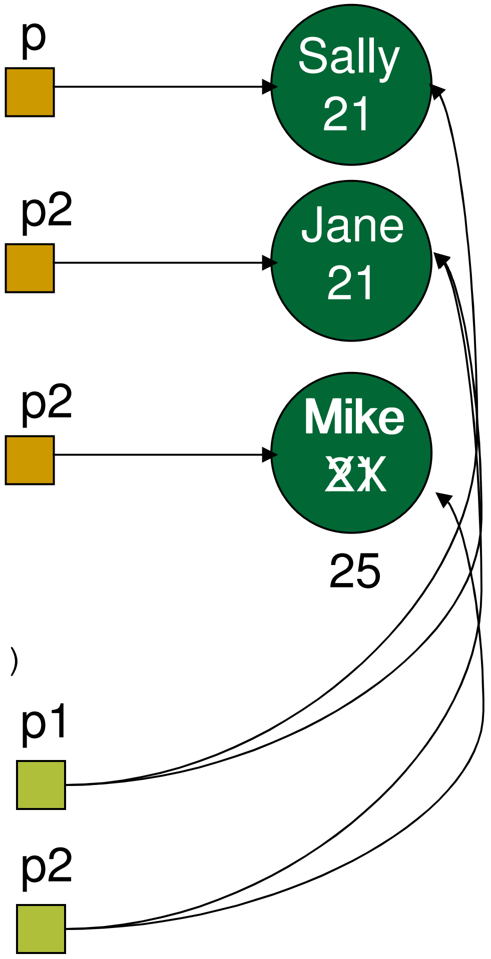
Testing if Object is a Person

Casting Object to a Person

End of class… no semicolon

Example Driver Program

```
public class PersonTest
{
   public static void main(String args[])
   {
        Person p = new Person(“Sally”);
        Person p2 = new Person(“Jane”);
        Person p3 = new Person(“Mike”);
        p3.setAge(25);
        PersonTest.compare(p, p2);
        compare(p2,p3);
   }

   public static void compare(Person p1, Person p2)
   {
        System.out.println(p1 + “ is “ +
                (p1.equals(p2)? “”: “not”) +
                “ the same age as “ + p2);
   }
}
```

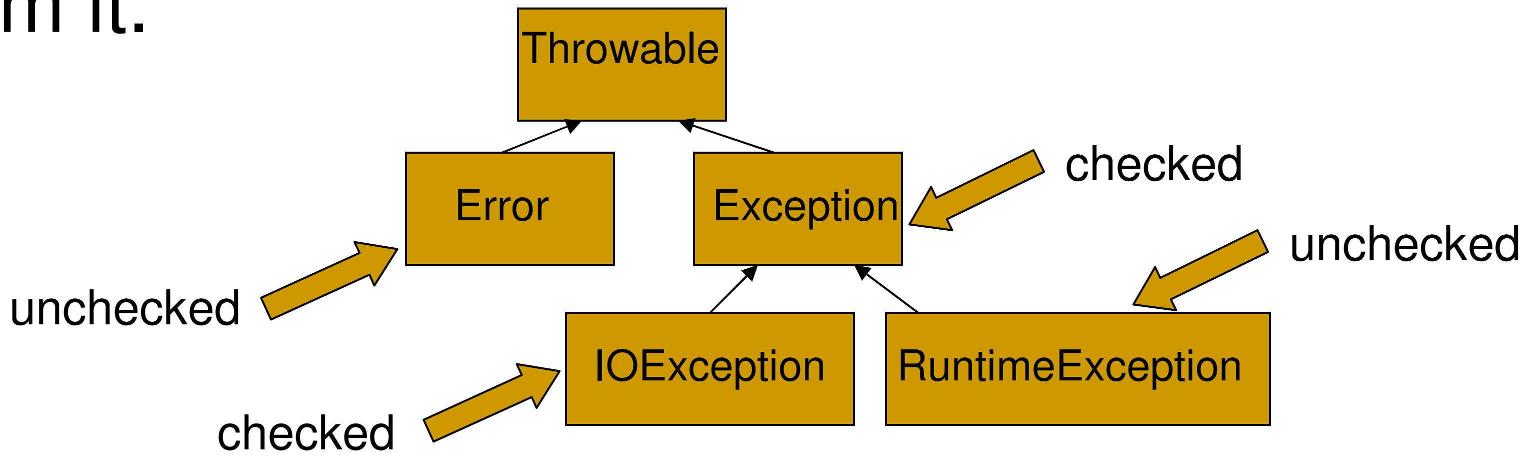


Exceptions

- Java handles exceptions like C++.
  - Place try block around problem code and a catch block immediately following try block to handle exceptions.
- Different from C++…
  - Java uses a finally block for code that is to be executed whether or not an exception is thrown.
  - Java has a built-in inheritance hierarchy for its exception classes which determines whether an exception is a checked or an unchecked exception.
  - You may declare that a method throws an exception to handle it. The exception is then passed up the call stack.
  - Java forces the programmer to handle a checked exception at compile time.

Exception Hierarchy

- Unchecked exceptions are derived from RuntimeException. Checked exceptions are derived from Exception. Error are also unchecked exceptions, but may not derive from it.



Handling the Exception Example

```
public class HandleExample
{
   public static void main(String args[])
   {
       try {
             String name = args[0];
             System.out.println(args[0]);
       } catch (IndexOutOfBoundsException e){
             System.out.println(“Please enter name ” +
                     “after java HandleExample”);
       } finally {
             System.out.println(“Prints no matter what”);
       }
   }
}
```

Passing up the Exception

- In Java you may pass the handling of the exception up the calling stack by declaring that the method throws the exception using the keyword `throws`.
- This is necessary for compilation if you call a method that throws a **checked exception** such as the `Thread.sleep` method.
- The Java API lists the exceptions that a method may throw. You may see the inheritance hierarchy of an exception in the API to determine if it is checked or unchecked.

Passing up the Exception Example

```
public class PassUpExample
{
   public static void main(String [] args){
        System.out.println(“Hello”);
        try
        {
               passback();
        }catch(InterruptedException e)
        {
               System.out.println(“Caught InterruptedException”);
        }
        System.out.println(“Goodbye”);
   }
   public static void passback() throws InterruptedException
   {
        Thread.sleep(3000);
   }
}
```

`main` is obligated to handle the exception since it is a checked exception

This method passes exception up call stack

This method throws a checked exception