Lecture Outline

- Why program defensively?
- Encapsulation
- Access Restrictions
- Unchecked Exceptions
- Checked Exceptions
DEFENSIVE PROGRAMMING
Why Program Defensively?

• Normally, your classes will form part of a larger system
• So other programmers will be using and relying upon your classes
• Obviously, your classes should be correct, but equally importantly, your classes should be robust – that is, resistant to accidental misuse by other programmers
• You should aim to ensure that no errors in the final system can be attributed to the behaviour of your classes
• We use the terminology “client code” for the code written by other programmers that is using your classes
Encapsulation

• One of the most important features of OOP is that it facilitates *encapsulation* – a class encapsulates both the data it uses, and the methods to manipulate the data

• The external user *only* sees the public methods of the class, and interacts with the objects of that class purely by calling those methods

• This has several benefits
  • Users are insulated from needing to learn details outside their scope of competence
  • Programmers can alter or improve the implementation without affecting any client code
Access Restrictions

• Encapsulation is enforced by the correct use of the access modifiers, public, private, <default>, and protected

• If you omit the access modifier, then you get the default, sometimes known as “package”

• These latter two modifiers are only really relevant for multi-package programs that use inheritance, so we need only consider public and private at the moment
public and private

- If an **instance variable** is public, then
  - Any object can *access* it directly
  - Any object can *alter* it directly
- If an **instance variable** is private, then
  - Objects that belong to *the same class* can access and alter it
  - Notice that privacy is a per-class attribute not per-object
- If a **method** is public, then
  - Any object can call that method
- If a **method** is private, then
  - Objects that belong to *the same class* can call it
Public Methods

- The *public interface* of a class is its list of public methods, which details all of the services that this class provides.
- Once a class is released (for example, as part of a library), then it is impossible or very difficult to change its public interface, because client code may use any of the public methods.
- Public methods must be precisely documented and robust to incorrect input and accidental misuse.
- Classes should make as *few* methods public as possible – limit them to just the methods needed for the class to perform its stated function.
Public variables

• Normally instance variables should not be public, since if client code can alter the values of instance variables then the benefit of encapsulation is lost.

• If client access to instance variables is desirable, then it should be provided by accessor and/or mutator methods (getters and setters).

• Advantages:
  • Maintenance of object integrity
  • Permits change of implementation
Simple Example

class MyDate {
    public int day;
    public String month;
    public int year;
}

MyDate md = new MyDate();
md.day = 31;
md.month = "Feb";

md is corrupt and so could cause problems elsewhere in the system
Use mutators instead

```java
public void setDay(int day) {
    // Check that day is valid for this.month
    // before setting the variables
}
public int getDay() {
    return this.day;
}
```

Setter methods act as “gatekeepers” to protect the integrity of objects.
Setters reject values that would create a corrupt object.
Getters return a value for client code to use, but do not allow the object itself to be changed.
JAVA EXCEPTIONS
Dealing with Errors

- Even if your classes are well-protected, errors still occur
  - Client code attempts to use your methods incorrectly, by passing incorrect or invalid parameter values
  - Your code cannot perform the services it is meant to due to circumstances outside your control (such as an Internet site being unavailable)
  - Your own code behaves incorrectly and/or your objects become corrupted
- Java provides *exceptions* (checked and unchecked) to handle these situations
Invalid Parameters

• The `String` method `charAt(int index)` returns the character at position index in a `String`.

• The only *valid* values for the parameter are numbers from 0 up to one less than the length of the `String`.

• What happens if `charAt(-1)` is ever called?
The method “throws” an exception

• If a parameter is invalid, then the method cannot do anything sensible with the request and so it creates an object from an Exception class and “throws” it

• If an Exception is thrown, then the runtime environment immediately tries to deal with it
  • If it is an unchecked exception, it simply causes the runtime to halt with an error message
  • If it is a checked exception, then the runtime tries to find some object willing to deal with it

• The method `charAt` throws a `StringIndexOutOfBoundsException` which is unchecked and hence causes the program to cease execution (crash!)
Throw your own exceptions

- Your own methods and/or constructors can throw exceptions if clients attempt to call them incorrectly
- This is how your code can enforce rules about how methods should be used
- For example, we can insist that the deposit and withdraw methods from the BankAccount class are called with positive values for the amount
- The general mechanism is to check the parameters and if they are invalid in some way to then
  - Create an object from class IllegalArgumentException
  - Throw that object
Throw your own

```java
public BankAccount(int amount) {
    if (amount >= 0) {
        balance = amount;
    } else {
        throw new IllegalArgumentException(
            "Account opening balance " +
            amount + " must be >0");
    }
}
```

- If the amount is negative then *declare* the variable ie, *create* the object and then *throw* it
- The constructor for IllegalArgumentException takes a String argument which is used for an error message that is returned to the user
- Throwing an exception is often used by *constructors* to prohibit the construction of invalid objects
“Predictable” errors

• **Unchecked** exceptions terminate program execution and are used when the client code must be seriously wrong.

• However, there are error situations that do not necessarily mean that the client code is incorrect, but reflect either a transient, predictable or easily-correctable mistake – this is *particularly* common when handling end-user input, or dealing with the operating system.

• For example, printers may be out of paper, disks may be full, Web sites may be inaccessible, filenames might be mistyped and so on.
Checked Exceptions

• Methods prone to such errors may elect to throw checked exceptions, rather than unchecked exceptions.

• Using checked exceptions is more complicated than using unchecked exceptions in two ways:
  • The programmer must declare that the method might throw a checked exception, and
  • The client code using that method is required to provide code that will be run if the method does throw an exception.
Client Perspective

• Many of the Java library classes declare that they *might* throw a checked exception

```java
public FileReader(File file) throws FileNotFoundException
Creates a new FileReader, given the File to read from.

Parameters:
    file - the File to read from

Throws:
    FileNotFoundException - if the file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.
```
try and catch

• If code *uses* a method that might throw a checked exception, then it *must* enclose it in a try/catch block

```java
try {
    FileReader fr = new FileReader("lect.ppt");
    // code for when everything is OK
} catch (java.io.FileNotFoundException e) {
    // code for when things go wrong
}
```

• *Try* to open and process this file, but *be prepared* to *catch* an exception if necessary
try and catch continued

• If everything goes smoothly, the code in the try block is executed, the code in the catch block is skipped.

• Otherwise, if one of the statements in the try block causes an exception to be thrown, then execution immediately jumps to the catch block, which tries to recover from the problem.

• What can the catch block do?
  • For human users: report the error and ask the user to change their request, or retype their password, or …
  • In all cases: Provide some feedback as to the likely cause of the error and how it may be overcome, even if it ultimately just causes execution to cease.
Using and Testing exceptions

```java
@Test(expected =
    IllegalArgumentException.class)
public void testIllegalDeposit() {
    BankAccount(-20);
}
```

- Java provides a many exception classes that cover most common possibilities
- Exceptions are simply objects in a Java program, so you can write your own classes of exceptions if desired
Some useful Java Exceptions

- `IllegalArgumentException`
- `IndexOutOfBoundsException`
- `NullPointerException`
- `ArithmeticException`
- `IOException`, `FileNotFoundException`
Exception vs RuntimeException

- Checked exceptions in Java extend the java.lang.Exception class
- Unchecked exceptions extend the java.lang.RuntimeException class
Programmer Perspective

• If you choose to write a method that throws a checked exception, then this must be declared in the source code, where you must specify the type of exception that might be thrown
  public void printFile(String fileName) throws java.io.FileNotFoundException {
    // Code that attempts to print the file
  }

• If you declare that your method might throw a checked exception, then the compiler will force any client code that uses your method to use a try/catch block

• This explicitly makes the client code responsible for these situations
Checked or Unchecked?

**Unchecked Exceptions**
- Any method can throw them without declaring the possibility
- No need for client code to use try/catch
- Causes execution to cease
- Used for fatal errors that are unexpected and unlikely to be recoverable

**Checked Exceptions**
- Methods must declare that they might throw them
- Client code must use try/catch
- Causes control flow to move to the catch block
- Used for situations that are not entirely unexpected and from which clients may be able to recover
- Use only if you think the client code might be able to do something about the problem
Summary

- Programming defensively means making your code robust to unexpected use.

- Use the need to know principle: Only expose the parts of your class that your client classes need to know.

- Java exceptions provide a uniform way of handling errors.

- Exceptions may be Unchecked or Checked.