# Data Structures

## OOP and Class Hierarchies

CS284

# Method `Object.equals`

- `Object.equals` method has a parameter of type Object

  ```
  public boolean equals (Object other) {...}
  ```

- Compares two objects to determine if they are equal
- A class must override equals in order to support comparison

# Employee.equals()

```
/** Determines whether the current object matches its
argument.
    @param obj The object to be compared to the current
    object;
    @return true if the objects have the same name and
    address; otherwise, return false
*/
@Override
public boolean equals(Object obj) {
    if (obj == this) return true;
    if (obj == null) return false;
    if (this.getClass() == obj.getClass()) {
        Employee other = (Employee) obj;
        return name.equals(other.name) &&
              address.equals(other.address);
    } else {
        return false;
    }
}
```

# Method `getClass`

- Every class *has a* Class object (that is created automatically when the class is loaded into an application)
- Method `getClass()` returns a reference to this unique object

```
Employee employee = new Employee();
System.out.println(employee.getClass());

// class Employee

Object employee = new Employee();
System.out.println(employee.getClass());

// class Employee
```

# Incompatible Types

- The following code generates a syntax error:

```
Object num_1 = new Integer(25);

Integer num_2 = num_1;
```

# Casting in a Class Hierarchy

- Casting obtains a reference of a different, but matching, type
- Casting does not change the object! It creates an anonymous reference to the object

```
Integer aNum = (Integer) aThing;
```

- The following line will work:

```
((Integer) aThing).intValue()
```

# Casting in a Class Hierarchy (cont.)

- Upcast:
  - Always valid but unnecessary
- Downcast:
  - Cast superclass type to subclass type
  - Java checks at run time to make sure it's legal
  - If it's not legal, it throws ClassCastException
- Question: when is a downcast legal?
  - Only when instantiated as a subclass object
  - Demo

# Using instanceof to Guard a Casting Operation

**`instanceof`** can guard against a `ClassCastException`

```java
Object obj = ...;
if (obj instanceof Integer) {
  Integer i = (Integer) obj;
  int val = i;
  ...;
} else {
  ...
}
```

# Polymorphism Eliminates Nested if Statements

```java
Number[] stuff = new Number[10];
// each element of stuff must reference actual
// object which is a subclass of Number
...

// Non OO style:
if (stuff[i] instanceof Integer)
  sum += ((Integer) stuff[i]).doubleValue();
else if (stuff[i] instanceof Double)
  sum += ((Double) stuff[i]).doubleValue();
...

// OO style:
sum += stuff[i].doubleValue();
```

# Polymorphism Eliminates Nested if Statements (cont.)

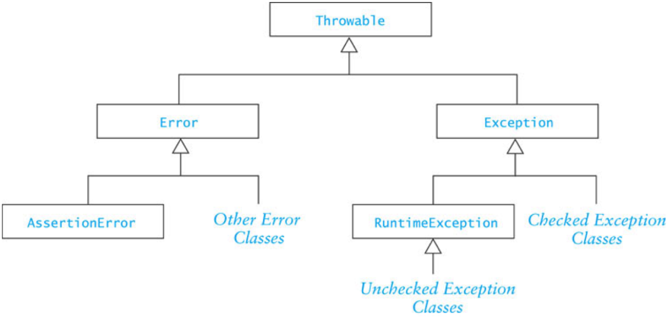
- Polymorphic code style is more extensible; it works automatically with new subclasses
- Polymorphic code is more efficient; the system does one indirect branch versus many tests
- Uses of instanceof may suggest poor coding style

# Run-time Errors or Exceptions

- Run-time errors
  - occur during program execution (i.e. at run-time)
  - occur when the JVM detects an operation that it knows to be incorrect
  - cause the JVM to throw an exception
- Examples of run-time errors include
  - division by zero
  - array index out of bounds
  - number format error
  - null pointer exception

# Class `Throwable`

- `Throwable` is the superclass of all exceptions
- All exception classes inherit its methods



# Checked and Unchecked Exceptions

- Checked exceptions
  - normally not due to programmer error
  - generally beyond the control of the programmer
  - all input/output errors are checked exceptions
  - Examples: IOException, FileNotFoundException
- Unchecked exceptions result from
  - programmer error (try to prevent them with defensive programming)
  - a serious external condition that is unrecoverable
  - Examples: NullPointerException, ArrayIndexOutOfBoundsException

# Checked Example

Suppose we type this code in order to prepare for reading from a text file...

```
File file = new File("file.txt");
BufferedReader reader = new BufferedReader(new FileReader(file));
```

Error: `Unhandled exception type FileNotFoundException`

# Unchecked Exceptions

- The class `Error` and its subclasses represent errors due to serious external conditions; they are unchecked
  - Example: OutOfMemoryError
  - You cannot foresee or guard against them
  - While you can attempt to handle them, it is generally not a good idea as you will probably be unsuccessful
- The class `Exception` and its subclasses can be handled by a program; they are also unchecked
  - `RuntimeException` and its subclasses are unchecked
  - All others must be either: explicitly caught or explicitly mentioned as thrown by the method

# Some Common Unchecked Exceptions

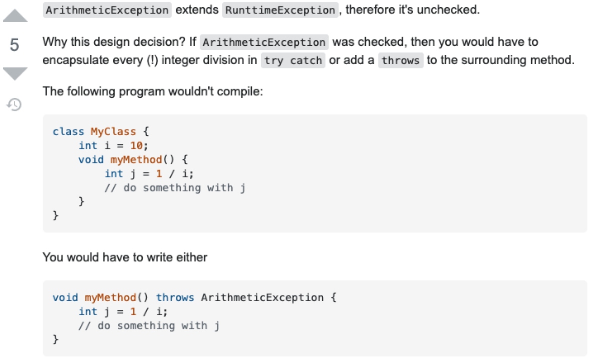
- `ArithmeticException`: division by zero, etc.
- `ArrayIndexOutOfBoundsException`
- `NumberFormatException`: converting a “bad” string to a number
- `NullPointerException`

```
@Override
public boolean equal (Shape s) {
         return this.area()==s.area();
}
```

What if `s` is null? Java does not force us to catch/throw `NullPointerException`

# Discussion

- Why are arithmetic exceptions unchecked?



`ArithmeticException` extends `RunttimeException`, therefore it's unchecked.

Why this design decision? If `ArithmeticException` was checked, then you would have to encapsulate every (!) integer division in `try catch` or add a `throws` to the surrounding method.

The following program wouldn't compile:

```java
class MyClass {
    int i = 10;
    void myMethod() {
        int j = 1 / i;
        // do something with j
    }
}
```

You would have to write either

```java
void myMethod() throws ArithmeticException {
    int j = 1 / i;
    // do something with j
}
```

- Why are null pointer exceptions unchecked?
- User defined exceptions are all *checked* exceptions

# Handling Exceptions

- When an exception is thrown, the normal sequence of execution is interrupted
- Default behavior (no handler)
  - Program stops
  - JVM displays an error message
- The programmer may provide a handle
  - Enclose statements in a **`try`** block
  - Process the exception in a **`catch`** block

# The try-catch Sequence

The try-catch sequence resembles an if-then-else statement

```
try {
  // Execute the following statements until an
  // exception is thrown
  ...
  // Skip the catch blocks if no exceptions were thrown
} catch (ExceptionTypeA ex) {
  // Execute this catch block if an exception of type
  // ExceptionTypeA was thrown in the try block
  ...
} catch (ExceptionTypeB ex) {
  // Execute this catch block if an exception of type
  // ExceptionTypeB was thrown in the try block
  ...
}
```

- `ExceptionTypeB` cannot be a subclass of `ExceptionTypeA`. If is was, its exceptions would be caught be the first catch clause and its catch clause would be unreachable.

# Using try-catch

User input is a common source of exceptions

```
public static int getIntValue(Scanner scan) {
  int nextInt = 0;          // next int value
  boolean validInt = false; // flag for valid input
  while(!validInt) {
    try {
      System.out.println("Enter number of kids: ");
      nextInt = scan.nextInt();
      validInt = true;
    } catch (InputMismatchException ex) {
      scan.nextLine();   // clear buffer
      System.out.println("Bad data-enter an integer");
    }
  }
  return nextInt;
}
```

# Throwing an Exception When Recovery is Not Obvious

- In some cases, you may be able to write code that detects certain types of errors, but there may not be an obvious way to recover from them
- In these cases an the exception can be thrown
- The calling method receives the thrown exception and must handle it

# Throwing an Exception When Recovery is Not Obvious (cont.)

```
public static void processPositiveInteger(int n) {
 if (n < 0) {
   throw new IllegalArgumentException("Invalid argument");
 } else {
   // Process n as required
   ...
 }
}
```

# Throwing an Exception When Recovery is Not Obvious (cont.)

A brief side comment: `IllegalArgumentException`, above, is unchecked. The following would not be accepted by Java

```
public static void processPositiveInteger(int n) {
  ... {
   throw new IOException("Invalid’’);
  }
}
```

We would have to write

```
public static void processPositiveInteger(int n)
throws IOException {
  ... {
   throw new IOException("Invalid’’);
  }
}
```

# Throwing an Exception When Recovery is Not Obvious (cont.)

```java
public static void main(String[] args) {
  Scanner scan = new Scanner(System.in);
  try {
    int num = getIntValue(scan);
    processPositiveInteger(num);
  } catch (IllegalArguementException ex) {
    System.err.println(ex.getMessage());
    System.exit(1);  // error indication
  }
  System.exit(0);  // normal exit
}
```

# Packages and Visibility

- A Java package is a group of cooperating classes
- The Java API is organized as packages
- Indicate the package of a class at the top of the file:

  ```
  package classPackage;
  ```
- Classes in the same package should be in the same directory (folder)
- The folder must have the same name as the package
- Classes in the same folder must be in the same package

# Packages and Visibility

- Classes not part of a package can only access public members of classes in the package
- If a class is not part of the package, it must access the public classes by their complete name, which would be `packagename.className`
- For example, `x = Java.awt.Color.GREEN;`
- If the package is imported, the packageName prefix is not required.

```
import java.awt.Color;
...
x = Color.GREEN;
```

# The Default Package

- Files which do not specify a package are part of the default package
- If you do not declare packages, all of your classes belong to the default package
- The default package is intended for use during the early stages of implementation or for small prototypes
- When you develop an application, declare its classes to be in the same package

# Visibility

- We have seen three visibility layers, public, protected, private
- A fourth layer, package visibility, lies between private and protected
- Classes, data fields, and methods with package visibility are accessible to all other methods of the same package, but are not accessible to methods outside the package
- Classes, data fields, and methods that are declared protected are visible within subclasses that are declared outside the package (in addition to being visible to all members inside the package)
- There is no keyword to indicate package visibility
- Package visibility is the default in a package if public, protected, private are not used

# Java Encapsulation

- The mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit
  - Variables are hidden from other classes
  - They can be accessed only through the methods of their current class
  - Also known as *data hiding*

# Visibility Supports Encapsulation

- Visibility rules enforce encapsulation in Java
  - private: for members that should be invisible even in subclasses
  - package: shields classes and members from classes outside the package
  - protected: provides visibility to extenders or classes in the package
  - public: provides visibility to all
- Encapsulation insulates against change: greater visibility means less encapsulation
- So use the most restrictive visibility possible to get the job done!