# CMPSCI 187: Programming With Data Structures

Lecture 8: Polymorphism, Generics, and Exceptions
23 September 2011

# Polymorphism, Generics, and Exceptions

- Object Polymorphism as in the Dog/Terrier Example
- Method Polymorphism Through Supertypes
- Why Not Always Use `Object`?
- Defining and Using Generic Types
- Exceptions and How They Can be Dealt With
- The Exceptions Class Hierarchy

# Object Polymorphism: The Dog/Terrier Example

- A **polymorphic reference** in Java code is a use of a variable in such a way the the behavior at run-time can depend on the actual class of the variable value.

- In general **polymorphism** (“many shapes”) refers to any programming language concept that can deal with data of different kinds.

- In our example where the Dog and Terrier classes each have their own `bark ( )` method, any reference `x.bark( ),` where `x` is a Dog variable, is polymorphic because the method invoked depends on the class of `x`’s value at run time.

# Method Polymorphism Through Supertypes

- In an example where different kinds of Dogs bark in different ways, we can define a method in the Dog class that gets overridden as needed. This works because there is a single supertype of all the classes that need to bark.
- So we can call the bark method from a Dog variable and the right method will run.
- But this isn't always satisfactory -- what if we had a method that was appropriate for some Dogs and not others, and the classes that were good for it did not form a single subclass of Dog.
- We could define a dummy method in the Dog class, that caused an exception if it were ever called. This should work because we only want to call it for the good Dogs. But this is not elegant and seems likely to lead to errors.

## Why Not Always Use `Object`?

- In a sense this unsatisfactory method is what we saw with the toString method of the class Object. Most classes need to have a toString method, and the desired String for an object depends entirely on the class. When we don't define the toString method, we get the default one.
- Suppose we have several different classes, unrelated in the class hierarchy, that all need to run the same method polymorphically.
- They aren't entirely unrelated, because they are all subclasses of Object.
- But we can't add another method to the Object class ourselves, because the code for the Object class is a fixed part of the Java language.
- We need a "fake superclass" of our diverse classes.

# Polymorphism With Interfaces

- An interface lies, in our diagram, above each of the classes implementing it.
- So it is in a sense a common “ancestor” of those classes.
- Suppose we have classes Human, Horse, and Train that each need to implement a `move` method in its own way, but those three classes are unrelated in the class hierarchy.
- We can define an interface Movable and have each of the three classes implement it.
- Then a variable of type Movable can have its move method called, polymorphically, and the correct code will be bound at run-time for the object. Now we can’t possibly call the method from an object that doesn’t have it.

# Defining and Using Generic Types

- We've seen the example of a generic class, `Stack<T>`, and a generic interface, L&S's `StackADT<T>`. Each makes a class, or interface, out of each possible class `T`.

- To write the code for a generic class or interface, we just put a class variable in angle brackets in the name of the class (T is traditional), and use T in the body as if it were a defined class.

- As we've seen, we use a particular one of these classes by referring, for example, to `Stack<Cell>`. It's as if we had a class written that replaced all the T's in the body of `Stack<T>` with Cells.

- We can restrict T in the generic class definition, by replacing `<T>` with, for example, `<T extends Dog>` or `<T implements Comparable>`.

# Exceptions and How They Can Be Dealt With

- We've seen exceptions in our projects, particularly during debugging. Most have been **run-time errors**, mistakes that crash our execution. These generally went away when we fixed the programming errors that caused them.

- But exceptions could be caused by user behavior even if our code is correct on normal input. We'd like to anticipate and deal with this, without just having the exception crash the program.

- Exceptions can be created by Java's code or our own -- we can say `if (thisHappens) throw new SomeKindOfException("Uh Oh");`

- There are two ways to deal with an exception -- we can **catch** it or **throw** it to another method where it is caught and dealt with.

# Throwing and Catching Exceptions

- If we have a some code in which we anticipate an exception that we can deal with, we can enclose it in a `try` block.
- If this `try` block is followed by one or more `catch` blocks, then at run time the interpreter sees whether the `try` block generates an exception that matches the exception variable declared in one of the `catch` blocks. If the `catch` block thus **catches** the exception, the interpreter runs the code in the first block for which this is true. Otherwise the exception is not caught there.
- We can put a `finally` block after the `try` and `catch` blocks, and it will be whatever happens. This is usually to clean up input/output objects.
- An uncaught exception crashes the execution unless it is thrown to the calling method by a `throws` clause. If the method call is in a `try` block, it could be caught in that method, or thrown further.

# The Exceptions Class Hierarchy

- Exceptions are Objects, of a particular subclass called Throwable, and they have their own hierarchy of subclasses. (See the diagram on page 518 of L&C, or the more complete list on page 64 of *Java Precisely*.)

- An exception is caught if its class fits into the type of the exception variable of the `catch` block.

- A **checked exception** is one such that any method that might produce one must have a **throws** clause in its declaration, matching (or "covering") any exception that could be thrown. Checked exceptions mostly involve file I/O, which is one reason that intro Java instructors try to avoid file I/O.

- You can declare your own subtypes of Exception, for behavior you anticipate and can deal with. They can be checked or unchecked depending on what they extend.