Polymorphism

- Another feature of OOP
- literally “having many forms”
- object variables in Java are polymorphic
- object variables can refer to objects or their declared type AND any objects that are descendants of the declared type

```
Property p = new Property();
p = new Railroad(); // legal!
p = new Utility();  // legal!
p = new Street();
Object obj1; // = what?
```

Data Type

- object variables have:
  - a **declared type**. Also called the static type.
  - a **dynamic type**. What is the actual type of the pointee at run time or when a particular statement is executed.

- Method calls are syntactically legal if the method is in the declared type or any ancestor of the declared type

- **The actual method that is executed at runtime is based on the dynamic type**
  - dynamic dispatch

Clicker Question 1

Consider the following class declarations:

```java
public class BoardSpace
public class Property extends BoardSpace
public class Street extends Property
public class Railroad extends Property
```

Which of the following statements would cause a syntax error? (Assume all classes have a default constructor.)

A. Object obj = new Railroad();
B. Street s = new BoardSpace();
C. BoardSpace b = new Street();
D. Railroad r = new Street();
E. More than one of these
Method LookUp

- To determine if a method is legal the compiler looks in the class based on the declared type
  - if it finds it great, if not go to the super class and look there
  - continue until the method is found, or the Object class is reached and the method was never found, (Compile error)
- To determine which method is actually executed the run time system:
  - starts with the actual run time class of the object that is calling the method
  - search the class for that method
  - if found, execute it, otherwise go to the super class and keep looking
  - repeat until a version is found
- Is it possible the runtime system won't find a method?

Clicker Question 2

What is output by the code to the right when run?
A. !!live
B. !eggegg
C. !egglive
D. !!!!
E. eggegglive

public class Animal{
    public String bt(){ return "!"; } }

public class Mammal extends Animal{
    public String bt(){ return "live"; } }

public class Platypus extends Mammal{
    public String bt(){ return "egg"; } }

Animal a1 = new Animal();
Animal a2 = new Platypus();
Mammal m1 = new Platypus();
System.out.print( a1.bt() );
System.out.print( a2.bt() );
System.out.print( m1.bt() );

Why Bother?

- Inheritance allows programs to model relationships in the real world
  - if the program follows the model it may be easier to write
- Inheritance allows code reuse
  - complete programs faster (especially large programs)
- Polymorphism allows code reuse in another way
- Inheritance and polymorphism allow programmers to create generic algorithms

Genericity

- One of the goals of OOP is the support of code reuse to allow more efficient program development
- If a algorithm is essentially the same, but the code would vary based on the data type
genericity allows only a single version of that code to exist
- in Java, there are 2 ways of doing this
  1. polymorphism and the inheritance requirement
  2. generics
A Generic List Class

Back to IntList

- We may find IntList useful, but what if we want a List of Strings? Rectangles? Lists?
  - What if I am not sure?
- Are the List algorithms different if I am storing Strings instead of ints?
- How can we make a generic List class?

Generic List Class

- required changes
- How does toString have to change?
  - why?!?
  - A good example of why keyword this is necessary from toString
- What can a List hold now?
- How many List classes do I need?

Writing an equals Method

- How to check if two objects are equal?
  if(objA == objA)
  // does this work?
- Why not this
  public boolean equals(List other)
- Because
  public void foo(List a, Object b)
  if( a.equals(b) )
    System.out.println( same )
  - what if b is really a List?
equals method

- read the javadoc carefully!
- Must handle null
- Parameter must be Object
  - otherwise overloading instead of overriding
  - causes
- must handle cases when parameter is not same data type as calling object
  - instanceof or getClass()
- don't rely on toString and String's equals

createASet examples

String[] sList = {"Texas", "texas", "Texas", "Texas", "UT", "texas"};
Object[] sSet = createASet(sList);
for(int i = 0; i < sSet.length; i++)
  System.out.println( sSet[i] );

Object[] list = {"Hi", 1, 4, 3.3, true,
                 new ArrayList(), "Hi", 3.3, 4};
Object[] set = createASet(list);
for(int i = 0; i < set.length; i++)
  System.out.println( set[i] );