Announcement

- Project 8 due Wed, April 29 at 10 pm.
- All questions on the class newsgroup.
Generics Motivation

- In Java, array elements must all be of the same type:
  - int[] counts = new int[10];
- Hence, arrays are type safe: The compiler will not let you put the wrong kind of thing into an array
- A collection, such as a Vector or the non-parameterized ArrayList, cannot hold primitives, but will accept any type of Object:
  - ArrayList someStuff = new ArrayList();
    someStuff.add("A String is an Object");
    someStuff.add(10);
- Is not type safe; Making a collection type safe is a tedious process
Generics

- J2SE 5.0 provides compile-time type safety with the Java Collections framework through generics.

- Generics allows you to specify, at compile-time, the types of objects you want to store in a Collection. Then when you add and get items from the list, the list already knows what types of objects are supposed to be acted on.

- So you don't need to cast anything. The "<>" characters are used to designate what type is to be stored. If the wrong type of data is provided, a compile-time exception is thrown.
Problem: Develop a Roster

- Purdue need rosters for undergraduate students, graduate students and employees.
- Each department needs rosters for their students and employees. Suppose there are ten departments.
- How many roster classes do we need to write?
Raw Solution

- The simplest solution is:
  - Write a class Roster which accept Objects.
  - We can have three objects: `undergraduateRoster`, `graduateRoster` and `employeeRoster`.
  - We can manually make sure correct objects are put in correct rosters.
  - When we get data from a roster, type casting is needed.

- If someone add an Integer to `graduateRoster` by mistake, what happens?
Inheritance Solution

- An intuitive solution after learning inheritance is like this:
  - Write a super class Roster.
  - Write three subclasses UndergraduateRoster, GraduateRoster and EmployeeRoster.
  - In EmployeeRoster, Student objects are not accepted.

- It looks reasonable, but what happens if a new roster like TARoster is needed? What happens if ten more different rosters are needed?
Generics Solution

- Now we use generics to solve it.
  - Write a class Roster<T>.
  - If we want a roster of undergraduate students, we can use Roster<UndergraduateStudent>.

- What’s the benefit here?
  - Easy to add new rosters.
  - Only need one class.
  - Do not accept incompatible objects.
  - Still has flexibility: Roster<Student> can accept UndergraduateStudent and GraduateStudent objects.
Some Issues about It

- We can define a Roster<T extends Person> to make sure all rosters can do something with class Person.
- What’s the relationship between Roster<Student> and Roster<GraduateStudent>?
  - Roster<Student> studentRoster = new Roster<GraduateStudent>(); correct?
Problem not Solved Yet!

- Each department needs rosters for their students and employees.
- Do we need Roster<T1, T2>, T1 for student or employee and T2 for department?
- What do you think?
- Remember Roster<GraduateStudent, CS> has no relationship with Roster<GraduateStudent, CE>.
Generics, Inheritance and Interface

- A subclass of a generic class must be a generic class.
- The super class of a generic class may not be a generic class.
  - Object class is a good example.
- A generic interface can only be implemented by corresponding generic class.
- A generic class can implement non-generic interface.
Raw Types

- A generic class can be used to produce different classes like `Roster<T>` described previously.
- However, Java still allow the declaration of `Roster` objects.
  - `Roster rawRoster = new Roster();`
- This roster may contain any kind of objects like `Integer`, `Circle`, `Car`...
- If we have `Roster<T extends Person>`, can it still accept any kinds of objects?
  - No!
Quiz

- If you are required to deal with complex numbers (a + bi).
- If a and b should have the same type but they can be Integer, Long, Float or Double. How will you define your classes?
- If a and b may not have the same type, what would you do?
- No need to write a complete class definition.