Generics

* A generic algorithm can be applied to different types of data
* Classic example: sorting
* Using the same logic (the exact same method), we can sort:
  * a list of grades numerically
  * a list of dates chronologically
  * a list of people alphabetically by name
  * a list of people by id
* We can define complex secondary sort criteria
Generics rely on polymorphism

Two approaches to generics in Java:

- Using type compatibility rules with inheritance and interfaces
- Using direct language support

Java 5 introduced generics formally into the language in 2004

We need to understand both approaches
Generics via Inheritance

Consider the following method:

```java
public void doIt(Object obj) {
    // contents
}
```

We can pass any object into this method.

Inside the method we are restricted to `Object` methods, such as `toString`.

Remember, the type of the object determines which version of a method is executed.
The `instanceOf` Operator

* If appropriate, the `instanceOf` operator can be used to determine the type of an object

```java
if (obj instanceOf Person)
    ((Person)obj).setLastName("Brooks");
```

* The object reference can be cast so that the compiler is satisfied

* The compiler will allow a call to `setLastName` through a `Person` reference, but not an `Object` reference
Generics via Inheritance

* Often the Object class is too generic for our purposes
* Suppose Employee contains an abstract method called pay
Arrays of Objects

```java
Employee[] staff = new Employee[50];
staff[0] = new Hourly(...);
staff[1] = new Salary(...);
staff[2] = new Executive(...);
```

Must have the "same" element type.
Generics via Inheritance

* Now a method can be designed that pays all employees:

```java
public void payAll(Employee[] staff)
{
   for (Employee emp : staff)
      emp.pay();
}
```

* No cast is necessary because all employees can be paid

* The proper `pay` method is invoked for each type of employee
Generics via Interfaces

* In similar ways, an interface can be used to support generic processing
* Often used when inheritance relationships are not appropriate
* Example: A sort can be defined for anything that is `Comparable`
* Any `Comparable` class must define the `compareTo` method
* It's up to each class to determine the relative order of the objects involved
Generics via Interfaces

A generic method can be defined to sort an array of Comparable items:

```java
public void sortIt(Comparable[] list)
{
    // code that sorts the list
}
```

If Person implements Comparable, then an array of Person objects is an array of Comparable objects
ArrayList

* The **ArrayList** class existed before generics were formally added to Java

    ArrayList list = new ArrayList();

* It stores **Object** references

* For backwards compatibility, you may still use it this way, but you'll get warnings

* It now is defined to accept a generic type

    ArrayList<Book> list = new ArrayList<Book>();
Java 5 Generics

* For a *generic class*, the type is specified each time an object is instantiated:

```java
ArrayList<Book> list = new ArrayList<Book>();
```

* Generics provide *type safety* -- only objects compatible with `Book` can be added to this list.

* And only objects compatible with `Date` can be added to this one:

```java
ArrayList<Date> list2 = new ArrayList<Date>();
```
When writing a generic class, you specify a type parameter and use it as appropriate

```java
public class Item<T> {
    private T element;

    public T getElement() {
        return element;
    }

    public void setElement(T element) {
        this.element = element;
    }
}
```
Generic Interfaces

* Interfaces can also be declared as generic

* The **Comparable** interface is now generic:

```java
public interface Comparable<T>
{
    public int compareTo(T other);
}
```

* For example, the **String** class now implements `Comparable<String>`

* Before, comparing two different **Comparable** objects resulted in a run-time error

* Now that problem is caught during compilation
Generic Type Bounds

* You can put bounds on a generic type

```java
class ThisClass<T extends Book>
```

* The bounds may be generic

```java
class ThatClass<T extends Comparable<T>>
```

* Note that `extends` is used with interfaces
Covariance

* In Java, inheritance leads to compatible array types

* If `Square` extends `Shape`, then `Square[]` is type compatible with `Shape[]`

* This is known as a *covariant array type*, which can lead to strange run-time problems

* So when generics were introduced, covariance was not allowed

* That is, `ArrayList<Square>` is NOT type compatible with `ArrayList<Shape>`
Generic Wildcards

However, *wildcards* can be used to define compatibility

```java
public static double totalArea
    (ArrayList<? extends Shape> list)
{
    double total = 0;
    for (Shape s : list)
        total += s.area();
    return total;
}
```

This method will accept an `ArrayList<Square>` or an `ArrayList<Shape>`