Generics and Type Safety

CS 180
Sunil Prabhapkar
Department of Computer Science
Purdue University
Generics are a new feature introduced in Java 5.

The main goal of this new feature is to improve program reliability through increased type safety.

Type safety reduces the possibility of incorrect types for data through compile-time testing and enforcement of correct types.

Although Java is type safe, earlier versions were unable to enforce type safety for generic collections, such as those defined in the Java collections framework (JCF), e.g. List and Map.

Prior to Java 5, these collections were unable to define a homogeneous collection -- they stored any object of any class.
Example: List

- If we create a list object for storing Student objects, we have no way of ensuring that non-Student objects are not added to this list.
- Similarly, we cannot create an ArrayList object in which we can only store objects of a certain class.
- This is desirable for improving the reliability of programs and avoiding logical errors.
Generic Classes

- Generic classes (or Generics) provide a solution for this.
- Before we see how this is achieved, let us get familiar with the notion of generic classes.
- Consider the functionality of a bag. Suppose we want to have a bag class that allows us to store one object of some class.
  - E.g. we would like to have one bag object to store a Book; a second one to store a Laptop, …
- One solution is to implement a BookBag and a LaptopBag class, and any other specific type of bag classes.
  - Each of these would have exactly the same functionality but with a data member of different types.
class Bag{
    private Object content;
    public Bag(){
        store(null) ;
    }
    public Object retrieve(){
        return content;
    }
    public void store(Object item){
        content = item;
    }
}
This does not allow us to enforce the requirement that the bookBag object can only store Book objects, and laptopBag only store Laptop objects.
Multiple classes

- Another solution is to implement a `BookBag` and a `LaptopBag` class, and any other specific type of bag classes.
  - Each of these would have exactly the same functionality but with a data member of different types.

```java
class BookBag{
    private Book content;
    public BookBag(){
        store(null) ;
    }
    public Book retrieve(){
        return content;
    }
    public void store(Book item){
        content = item;
    }
}

class LaptopBag{
    private Laptop content;
    public LaptopBag(){
        store(null);
    }
    public Laptop retrieve(){
        return content;
    }
    public void store(Laptop item){
        content = item;
    }
}
```
Another solution is to implement a BookBag and a LaptopBag class, and any other specific type of bag classes.

- Each of these would have exactly the same functionality but with a data member of different types.

```java
class BookBag{
    private Book content;
    public BookBag(){
        store(null);
    }
    public Book retrieve(){
        return content;
    }
    public void store(Book item){
        content = item;
    }
}
class LaptopBag{
    private Laptop content;
    public LaptopBag(){
        store(null);
    }
    public Laptop retrieve(){
        return content;
    }
    public void store(Laptop item){
        content = item;
    }
}
```
Another solution is to implement a BookBag and a LaptopBag class, and any other specific type of bag classes.

- Each of these would have exactly the same functionality but with a data member of different types.
Both alternatives are unsatisfactory.

Instead, we can create a generic class (Java 5 onwards only) that solves both problems.

The generic class takes a Type parameter as part of its definition.
Generic class

class Bag<T>{
    private T content;
    public Bag(){
        store(null) ;
    }
    public T retrieve(){
        return content;
    }
    public void store(T item){
        content = item;
    }
}

Bag <Book> bookBag = new
    Bag <Book> ();
Bag <Laptop> laptopBag = new
    Bag <Laptop> ();
Book b = new Book();
Laptop l = new Laptop();

bookBag.store(b);
bookBag.store(l);
l = (Laptop)laptopBag.retrieve();
b = (Book)laptopBag.retrieve();
Generic class

class Bag<T>{
    private T content;
    public Bag(){
        store(null);
    }
    public T retrieve(){
        return content;
    }
    public void store(T item){
        content = item;
    }
}

Bag<Book> bookBag = new Bag<Book>();
Bag<Laptop> laptopBag = new Bag<Laptop>();
Book b = new Book();
Laptop l = new Laptop();
bookBag.store(b);
bookBag.store(l);
l = (Laptop)laptopBag.retrieve();
b = (Book)laptopBag.retrieve();
Generic class

Bag is called a generic (pluggable, parametrized) class.
T is called a type parameter

```java
class Bag<T>{
    private T content;
    public Bag(){
        store(null) ;
    }
    public T retrieve(){
        return content;
    }
    public void store(T item){
        content = item;
    }
}
```

Bag <Book> bookBag = new Bag <Book> ();
Bag <Laptop> laptopBag = new Bag <Laptop> ();
Book b = new Book();
Laptop l = new Laptop();

bookBag.store(b);
bookBag.store(l);
l = (Laptop)laptopBag.retrieve();
b = (Book)laptopBag.retrieve();
Bag is called a generic (pluggable, parametrized) class.

T is called a type parameter

```
class Bag<T>{
    private T content;
    public Bag(){
        store(null) ;
    }
    public T retrieve(){
        return content;
    }
    public void store(T item){
        content = item;
    }
}
```

```java
Bag <Book> bookBag = new Bag <Book> ();
Bag <Laptop> laptopBag = new Bag <Laptop> ();
Book b = new Book();
Laptop l = new Laptop();

bookBag.store(b);
bookBag.store(l);
l = (Laptop)laptopBag.retrieve();
b = (Book)laptopBag.retrieve();
```
Generic classes and objects

- Depending upon the type specified for the type parameter when creating an object of the Bag class, the compiler will perform the appropriate type checking.
  - The content of bookBag can only be an object of class Book (or descendants)
  - The content of laptopBag can only be an object of class Laptop (or descendants)

- This is exactly what we wanted
  - We define only a single class, and
  - We are able to limit the data types for different instances.
Generic classes

- It is important to realize that there is only one class called Bag.
- `bookBag` and `laptopBag` are both of class Bag.
  - Note the class is called Bag, the constructor is `Bag()`.
  - There are no `Bag<T>`, or `Bag<Book>`, … classes.
- However, even though they are from the same class, we **cannot** use them interchangeably:
  - `bookBag = laptopBag;`
Generic classes

Bag <Book> bookBag = new Bag <Book> ();
Bag <Laptop> laptopBag = new Bag <Laptop> ();

if(bookBag instanceof Bag)
   System.out.println("bookBag is a Bag object");

if(bookBag instanceof Bag<T>)
   System.out.println("bookBag is a Bag<T> object");

if(bookBag instanceof Bag<Book>)
   System.out.println("bookBag is a Bag<Book> object");

if(bookBag instanceof Bag<Laptop>)
   System.out.println("bookBag is a Bag<Laptop> object");
Generic classes

Bag <Book> bookBag = new Bag <Book> ();
Bag <Laptop> laptopBag = new Bag <Laptop> ();

if(bookBag instanceof Bag)
   System.out.println("bookBag is a Bag object");

if(bookBag instanceof Bag<T>)
   System.out.println("bookBag is a Bag<T> object");

if(bookBag instanceof Bag<Book>)
   System.out.println("bookBag is a Bag<Book> object");

if(bookBag instanceof Bag<Laptop>)
   System.out.println("bookBag is a Bag<Laptop> object");

Will not compile!
Generic classes

Bag <Book> bookBag = new Bag <Book> ();
Bag <Laptop> laptopBag = new Bag <Laptop> ();

if(bookBag instanceof Bag)
   System.out.println("bookBag is a Bag object");
if(bookBag instanceof Bag<T>)
   System.out.println("bookBag is a Bag<T> object");
if(bookBag instanceof Bag<Book>)
   System.out.println("bookBag is a Bag<Book> object");
if(bookBag instanceof Bag<Laptop>)
   System.out.println("bookBag is a Bag<Laptop> object");

Will not compile!
Generic classes

Bag <Book> bookBag = new Bag <Book> ();
Bag <Laptop> laptopBag = new Bag <Laptop> ();

if(bookBag instanceof Bag)
   System.out.println("bookBag is a Bag object");
if(bookBag instanceof Bag<T>)
   System.out.println("bookBag is a Bag<T> object");
if(bookBag instanceof Bag<Book>)
   System.out.println("bookBag is a Bag<Book> object");
if(bookBag instanceof Bag<Laptop>)
   System.out.println("bookBag is a Bag<Laptop> object");

Will not compile!
Primitive data

- A type parameter can only take a class data type.
- If we want to store primitive data in our Bag, then we have to convert it to its corresponding wrapper class.
- With Java 5 this is trivial due to autoboxing and auto unboxing.
Primitive data

- A type parameter can only take a class data type.
- If we want to store primitive data in our Bag, then we have to convert it to its corresponding wrapper class.
- With Java 5 this is trivial due to autoboxing and auto unboxing.

```java
Bag<Integer> intBag = new Bag<Integer>();
int i = 5, j;
intBag.store(new Integer(i));
intBag.store(i);
j = intBag.retrieve();
j = intBag.retrieve().intValue();
```
Multiple Type Parameters

- We can have multiple type parameters in a generic class.
Multiple Type Parameters

- We can have multiple type parameters in a generic class.
Example
Example

```java
DualBag<Book, Laptop> bigBag = new DualBag<Book, Laptop> ();
Laptop l = new Laptop();
Book b = new Book();
bigBag.storeFirst(b);
bigBag.storeSecond(l);

Book b1 = bigBag.retrieveFirst();
Laptop l1 = bigBag.retrieveSecond();
```
Example

```java
DualBag<Book, Laptop> bigBag = new DualBag<Book, Laptop> ();
Laptop l = new Laptop();
Book b = new Book();
bigBag.storeFirst(b);
bigBag.storeSecond(l);

Book b1 = bigBag.retrieveFirst();
Laptop l1 = bigBag.retrieveSecond();
b1 = bigBag.retrieveSecond();
```
Example

```java
DualBag<Book, Laptop> bigBag = new DualBag<Book, Laptop>();
Laptop l = new Laptop();
Book b = new Book();
bigBag.storeFirst(b);
bigBag.storeSecond(l);

Book b1 = bigBag.retrieveFirst();
Laptop l1 = bigBag.retrieveSecond();
b1 = bigBag.retrieveSecond();
```

Will not compile!
Example

```java
DualBag<Book, Laptop> bigBag = new DualBag<Book, Laptop> ();
Laptop l = new Laptop();
Book b = new Book();
bigBag.storeFirst(b);
bigBag.storeSecond(l);

Book b1 = bigBag.retrieveFirst();
Laptop l1 = bigBag.retrieveSecond();

b1 = bigBag.retrieveSecond();
l1 = bigBag.retrieveFirst();
```

Will not compile!
Example

```java
DualBag<Book, Laptop> bigBag = new DualBag<Book, Laptop> ();
Laptop l = new Laptop();
Book b = new Book();
bigBag.storeFirst(b);
bigBag.storeSecond(l);

Book b1 = bigBag.retrieveFirst();
Laptop l1 = bigBag.retrieveSecond();

b1 = bigBag.retrieveSecond();
l1 = bigBag.retrieveFirst();
```

Will not compile!
Example

```java
DualBag<Book, Book> pair =
    new DualBag<Book, Book> ();
Book b1 = new Book();
Book b2 = new Book();
parr.storeFirst(b1);
pair.storeSecond(b2);
```
Both parameters can be of the same type.
Example

Both parameters can be of the same type.

If we want to enforce this then we use only one parameter.
Example

Both parameters can be of the same type.

If we want to enforce this then we use only one parameter.

class DualBag<T> {
    private T content1;
    private T content2;
    public DualBag() {
        storeFirst(null);
        storeSecond(null);
    }
    public DualBag(T item1, T item2) {
        storeFirst(item1);
        storeSecond(item2);
    }
    public T retrieveFirst() {
        return content1;
    }
    public T retrieveSecond() {
        return content2;
    }
    public void storeFirst(T item) {
        content1 = item;
    }
    public void storeSecond(T item) {
        content2 = item;
    }
}
Type Safety

These two are not the same.

- Only the generic class offers (compile time) type safety.
- The other may lead to runtime errors due to type mismatches.
BagA bookBag = new BagA();
BagA laptopBag = new BagA();
Laptop l = new Laptop();
Book b = new Book();

bookBag.store(l);
laptopBag.store(l);

l = (Laptop) bookBag.retrieve();
b = (Book) laptopBag.retrieve();
BagA bookBag = new BagA();
BagA laptopBag = new BagA();
Laptop l = new Laptop();
Book b = new Book();

bookBag.store(l);
laptopBag.store(l);

l = (Laptop) bookBag.retrieve();
b = (Book) laptopBag.retrieve();

Runtime error!
Type Safety contd.

BagA bookBag = new BagA();
BagA laptopBag = new BagA();
Laptop l = new Laptop();
Book b = new Book();

bookBag.store(l);
laptopBag.store(l);

l = (Laptop) bookBag.retrieve();
b = (Book) laptopBag.retrieve();

Runtime error!

BagB<Book> bookBag = new BagB<Book>();
BagB<Laptop> laptopBag = new BagB<Laptop>();
Laptop l = new Laptop();
Book b = new Book();

bookBag.store(l);
laptopBag.store(b);

l = (Laptop) bookBag.retrieve();
b = (Book) laptopBag.retrieve();
Type Safety contd.

BagA bookBag = new BagA();
BagA laptopBag = new BagA();
Laptop l = new Laptop();
Book b = new Book();

bookBag.store(l);
laptopBag.store(l);

l = (Laptop) bookBag.retrieve();
b = (Book) laptopBag.retrieve();

BagB<Book> bookBag = new BagB<Book>();
BagB<Laptop> laptopBag = new BagB<Laptop>();
Laptop l = new Laptop();
Book b = new Book();

bookBag.store(l);
laptopBag.store(b);

l = (Laptop) bookBag.retrieve();
b = (Book) laptopBag.retrieve();
Type Safety contd.

```java
BagA bookBag = new BagA();
BagA laptopBag = new BagA();
Laptop l = new Laptop();
Book b = new Book();

bookBag.store(l);
laptopBag.store(l);

l = (Laptop) bookBag.retrieve();
b = (Book) laptopBag.retrieve();
```

```java
BagB<Book> bookBag = new BagB<Book>();
BagB<Laptop> laptopBag = new BagB<Laptop>();
Laptop l = new Laptop();
Book b = new Book();

bookBag.store(l);
laptopBag.store(b);

l = (Laptop) bookBag.retrieve();
b = (Book) laptopBag.retrieve();
```

Runtime error!

Will not compile!
Type Safety contd.

BagA bookBag = new BagA();
BagA laptopBag = new BagA();
Laptop l = new Laptop();
Book b = new Book();

bookBag.store(l);
laptopBag.store(l);

l = (Laptop) bookBag.retrieve();
b = (Book) laptopBag.retrieve();

Runtime error!

BagB<Book> bookBag = new BagB<Book>();
BagB<Laptop> laptopBag = new BagB<Laptop>();
Laptop l = new Laptop();
Book b = new Book();

bookBag.store(l);
laptopBag.store(b);

l = (Laptop) bookBag.retrieve();
b = (Book) laptopBag.retrieve();

Will not compile!
BagA bookBag = new BagA();
BagA laptopBag = new BagA();
Laptop l = new Laptop();
Book b = new Book();

bookBag.store(l);
laptopBag.store(l);

l = (Laptop) bookBag.retrieve();
b = (Book) laptopBag.retrieve();
Type Safety

Which is better
- Have the compiler enforce type safety or
- Allow unsafe expressions and then have a runtime error?

It is far better to catch such errors (which are likely to be logical errors) at compile time. Thus type safety is desirable.

In addition to better reliability, note that with type safety there is no need for type casting or runtime type checking. This can improve the runtime of programs too.
Bounded Types

- The current generic version of Bag allows us to create instances to store any type of objects as content.
- What if we want to limit the types of objects that can be stored in our Bag?
- We can do this by bounding the type parameter.
- Consider a Bag class that only allows us to store numeric objects.
class NumBag<T extends Number>{
   private T content;
   public NumBag(){
      store(null);
   }
   public NumBag(T item){
      store(item);
   }
   public T retrieve(){
      return content;
   }
   public void store(T item){
      content = item;
   }
}
class NumBag<T extends Number>{
    private T content;
    public NumBag(){
        store(null);
    }
    public NumBag(T item){
        store(item);
    }
    public T retrieve(){
        return content;
    }
    public void store(T item){
        content = item;
    }
}
class NumBag<T extends Number> {
    private T content;
    public NumBag() {
        store(null);
    }
    public NumBag(T item) {
        store(item);
    }
    public T retrieve() {
        return content;
    }
    public void store(T item) {
        content = item;
    }
}

NumBag<Double> doubleBag = new NumBag<Double>();
NumBag<Double> laptopBag = new NumBag<Double>();
NumBag<Number> numBag;

doubleBag.store(new Double(3.0));
numBag.store(new Number(12));
numBag.store(new Integer(12));
numBag = new NumBag<Double>();
Bounded Type

```java
class NumBag<T extends Number>{
   private T content;
   public NumBag(){
      store(null) ;
   }
   public NumBag(T item){
      store(item) ;
   }
   public T retrieve(){
      return content;
   }
   public void store(T item){
      content = item;
   }
}

NumBag <Double> doubleBag = new NumBag <Double> ();
NumBag <Laptop> laptopBag = new NumBag <Laptop> ();
NumBag <Number> numBag;

doubleBag.store(new Double(3.0));
numBag.store(new Number(12));
numBag.store(new Integer(12));
numBag = new NumBag<Double>();
```

Laptop doesn’t extend Number

Will not compile!
Bounded Type

```java
class NumBag<T extends Number>{
   private T content;
   public NumBag(){
      store(null) ;
   }
   public T retrieve(){
      return content;
   }
   public void store(T item){
      content = item;
   }
}
```

```java
NumBag <Double> doubleBag = new NumBag <Double> ();
NumBag <Laptop> laptopBag = new NumBag <Laptop> ();
NumBag <Number> numBag;

doubleBag.store(new Double(3.0));
numBag.store(new Number(12));
numBag.store(new Integer(12));
umBag = new NumBag<Double>();
```

Number is an abstract class!

Will not compile!

Laptop doesn't extend Number
Number is an abstract class!
Bounded Type

```java
class NumBag<T extends Number> {
    private T content;
    public NumBag() {
        store(null);
    }
    public NumBag(T item) {
        store(item);
    }
    public T retrieve() {
        return content;
    }
    public void store(T item) {
        content = item;
    }
}
```

```java
NumBag<Double> doubleBag = new NumBag<Double>();
NumBag<Laptop> laptopBag = new NumBag<Laptop>();
NumBag<Number> numBag;

doubleBag.store(new Double(3.0));
umBag.store(new Number(12));
umBag.store(new Integer(12));
umBag = new NumBag<Double>();

Will not compile!

Laptop doesn't extend Number
Number is an abstract class!
No super-sub relationship!
```
Wildcard types

Suppose we would like to be able to check for equality of the number values stored in different instances of the NumBag class:

```java
NumBag<Double> bag1 = new NumBag<Double>(new Double(3.0));
NumBag<Integer> bag2 = new NumBag<Integer>(new Integer(3));
if(bag1.isSameValue(bag2))
    System.out.println("The values are the same.");
```

How should the isSameValue method be written?

Note that it may be called on an object such as `bag1` that stores doubles and receive an object such as `bag2` that stores integers as a parameter.
Wildcard types
class NumBag<T extends Number>{
   private T content;
   ... 
   public boolean isSameValue(NumBag<T> item){
      return this.retrieve().doubleValue() == item.retrieve().doubleValue();
   }
}
Wildcard types

```java
class NumBag<T extends Number>{
    private T content;
    . . .
    public boolean isSameValue(NumBag<T> item){
        return this.retrieve().doubleValue() ==
            item.retrieve().doubleValue();
    }
}
```

```
NumBag<Double> bag1= new NumBag<Double>(new Double(3.0));
NumBag<Integer> bag2 = new NumBag<Integer>(new Integer(3));
if(bag1.isSameValue(bag2))
    System.out.println("The values are the same.");
```

Will not compile!
**Wildcard types**

```java
class NumBag<T extends Number>{
   private T content;
   . . . 
   public boolean isSameValue(NumBag<?> item){
      return this.retrieve().doubleValue() == 
             item.retrieve().doubleValue();
   }
}
```
class NumBag<T extends Number>{
    private T content;
    
    public boolean isSameValue(NumBag<?> item){
        return this.retrieve().doubleValue() == item.retrieve().doubleValue();
    }
}

NumBag<Double> bag1= new NumBag<Double>(new Double(3.0));
NumBag<Integer> bag2 = new NumBag<Integer>(new Integer(3));

if(bag1.isSameValue(bag2))
   System.out.println("The values are the same.");
class NumBag<T extends Number> {
    private T content;
    . . .
    public boolean isSameValue(NumBag<? extends Number> item) {
        return this.retrieve().doubleValue() ==
        item.retrieve().doubleValue();
    }
}

Okay as long as bag2’s content supports the doubleValue() method

NumBag<Double> bag1 = new NumBag<Double>(new Double(3.0));
NumBag<Integer> bag2 = new NumBag<Integer>(new Integer(3));
if (bag1.isSameValue(bag2))
   System.out.println("The values are the same.");
Limitations of generics

- We are not allowed to
  - Create an instance of the type parameter class within the generic class, or
  - Use type parameters with static data members or static methods.

- The reason is that the actual type corresponding to the type parameter is known only when an object is instantiated.
Example

class NumBag<T extends Number> {
   private T content;
   private static T classContent;
   public NumBag() {
      content = new T();
      store(content);
   }
   public static T getClassContent() {
      return classContent;
   }
}
class NumBag<T extends Number> {
   private T content;
   private static T classContent;
   public NumBag() {
      content = new T();
      store(content);
   }
   public static T getClassContent() {
      return classContent;
   }
}

Will not compile!
Example

class NumBag<T extends Number> {
   private T content;
   private static T classContent;
   public NumBag() {
      content = new T();
      store(content);
   }
   public static T getClassContent() {
      return classContent;
   }
}

Will not compile!
Example

class NumBag<T extends Number> {
    private T content;
    private static T classContent;
    public NumBag() {
        content = new T();
        store(content);
    }
    public static T getClassContent() {
        return classContent;
    }
}

Will not compile!
Java Collections Framework

- The JCF provides a large set of very commonly used classes such as ArrayList, Vector, Queue, ...
- In earlier releases of Java the JCF classes were unable to enforce types strongly.
- All JCF classes have been retrofitted in Java 5 to work correctly with generics.
- These classes are defined in java.util
import java.util.*;

List bookList = new ArrayList();

booklist.add(new Book("Wilder");
booklist.add(new Book("Nabokov");
booklist.add(new Book("William");
bookList.add("Shakespeare");
import java.util.*;

List bookList = new ArrayList();

booklist.add(new Book("Wilder"));
booklist.add(new Book("Nabokov"));
booklist.add(new Book("William"));
bookList.add("Shakespeare");

Can add any type of object to this ArrayList.
(Old) JCF Example

```java
import java.util.*;

List bookList = new ArrayList();

booklist.add(new Book("Wilder"));
booklist.add(new Book("Nabokov"));
booklist.add(new Book("William"));
bookList.add("Shakespeare");

Iterator itr = bookList.iterator();
while(itr.hasNext()){
   Book book = (Book) itr.Next();
   System.out.println(book.getAuthor());
}
```

Can add any type of object to this ArrayList.
(Old) JCF Example

```java
import java.util.*;
List bookList = new ArrayList();
bookList.add(new Book("Wilder");
bookList.add(new Book("Nabokov");
bookList.add(new Book("William");
bookList.add("Shakespeare");

Iterator itr = bookList.iterator();
while(itr.hasNext()){
   Book book = (Book) itr.Next();
   System.out.println(book.getAuthor());
}
```

Can add any type of object to this ArrayList.

Note the need for typecasting.
import java.util.*;

ArrayList<Book> bookList = new ArrayList<Book>();

bookList.add(new Book("Wilder");
bookList.add(new Book("Nabokov");
bookList.add(new Book("William");

bookList.add("Shakespeare");
Generics JCF Example

```java
import java.util.*;

ArrayList<Book> bookList = new ArrayList<Book>();
bookList.add(new Book("Wilder");
bookList.add(new Book("Nabokov");
bookList.add(new Book("William");
bookList.add("Shakespeare");

Compile error.
```
Generics JCF Example

```java
import java.util.*;

ArrayList<Book> bookList = new ArrayList<Book>();

bookList.add(new Book("Wilder");
bookList.add(new Book("Nabokov");
bookList.add(new Book("William");

bookList.add("Shakespeare");

Iterator<Book> itr = bookList.iterator();
while(itr.hasNext()){
   Book book = itr.next();
   System.out.println(book.getAuthor());
}
```

Compile error.
Generics JCF Example

```java
import java.util.*;
ArrayList<Book> bookList = new ArrayList<Book>();

bookList.add(new Book("Wilder");
bookList.add(new Book("Nabokov");
bookList.add(new Book("William");
bookList.add("Shakespeare");

Iterator<Book> itr = bookList.iterator();
while(itr.hasNext()){  // Compile error.
   Book book = itr.Next();
   System.out.println(book.getAuthor());
}
```

Iterator needs type too.
import java.util.*;

ArrayList<Book> bookList = new ArrayList<Book>();

bookList.add(new Book("Wilder"));
bookList.add(new Book("Nabokov"));
bookList.add(new Book("William"));
bookList.add("Shakespeare");

Iterator<Book> itr = bookList.iterator();
while(itr.hasNext()){
    Book book = itr.Next();
    System.out.println(book.getAuthor());
}

Compile error.

No need for typecasting.

Iterator needs type too.
New *for* statement

- Java 5 has also introduced a new variant of the for loop statement: *for-each* loop
- Eliminates the need for iterator objects for iterating through a collection.
  ```java
  for (Book book : bookList) {
      System.out.println(book.getAuthor());
  }
  ```
- General format:
  ```java
  for (<type> <var> : <collection>) <stmt>
  ```
- Collection must be a JCF class and type must match the type of the collection.
for-each Example with ArrayList

```
ArrayList<Bag<Book>> bookBagList = new ArrayList<Bag<Book>>();
bookBagList.add(new Bag<Book>(new Book("Qian")));
bookBagList.add(new Bag<Book>(new Book("Achebe")));
bookBagList.add(new Bag<Book>(new Book("Lisa")));

for(Bag<Book> bag : bookBagList) {
   System.out.println(bag.retrieve().getAuthor());
}
```
for-each Example with arrays

```java
int number[] = {1, 3, 4, 5, 6, 9};

int sum = 0;
for (int element : number) {
    sum += element;
}

int sum = 0;
for (int i = 0; i < number.length; i++)
    {
    sum += number[i];
    }
```
**for-each** Example with arrays

```java
int number[] = {1, 3, 4, 5, 6 9};

int sum = 0;
for(int element : number) {
    sum += element;
}

int sum = 0;
for (int i = 0; i< number.length; i++)
{
    sum += number[i];
}
```

**NOTE:** not allowed to change the value of loop variable of a for-each loop.
Non-generic ArrayList example

```java
ArrayList bookList = new ArrayList();
bookList.add(new Book("Qian"));
bookList.add(new Book("Achebe"));
bookList.add(new Book("Lisa"));

for(Object b : bookBagList) {
    System.out.println( ((Book)b).getAuthor() );
}
```
Generics, Inheritance & Interfaces

- It is possible to define a generic subclass of a non-generic class.
- It is also possible to define a subclass of a generic class
  - IMPORTANT: in this case the subclass must also be generic.
- It is possible for a generic class to implement a non-generic interface.
- It is also possible for a class to implement a generic interface
  - IMPORTANT: in this case the class must also be generic.
Subclass of a generic class

class Bag<T>{
   private T content;
   public Bag(){
      store(null) ;
   }
   public T retrieve(){
      return content;
   }
   public void store(T item){
      content = item;
   }
}
Subclass of a generic class

class Bag<T>{
   private T content;
   public Bag(){
      store(null) ;
   }
   public T retrieve(){
      return content;
   }
   public void store(T item){
      content = item;
   }
}

class BagSub<T> extends Bag<T>{
   private int id;
   private static int idCounter = 100;
   public BagSub(){
      super();
      id = idCounter++;
      store(item) ;
   }
   public int showID(){
      return id;
   }
}
Subclass of a generic class

class Bag<T>{
   private T content;
   public Bag(){
      store(null);
   }
   public T retrieve(){
      return content;
   }
   public void store(T item){
      content = item;
   }
}

class BagSub<T> extends Bag<T>{
   private int id;
   private static int idCounter = 100;
   public BagSub(){
      this(null);
   }
   public BagSub(T item){
      super();
      id = idCounter++;
      store(item);
   }
   public int showID(){
      return id;
   }
}
Example use

BagSub<String> subStr;
BagSub<Integer> subInt;

subStr = new BagSub<String> ("Hello");
subInt = new BagSub<Integer>();

System.out.println(subStr.showID() + ": " + subStr.retrieve());
System.out.println(subInt.showID() + ": " + subInt.retrieve());
Example use

BagSub<String> subStr;
BagSub<Integer> subInt;

subStr = new BagSub<String> ("Hello");
subInt = new BagSub<Integer>( );

System.out.println(subStr.showID() + " : " + subStr.retrieve());
System.out.println(subInt.showID() + " : " + subInt.retrieve());

100: Hello
101: null
### Subclass of a generic class

```java
class Bag<T>{
    private T content;
    public Bag(){
        store(null);
    }
    public T retrieve(){
        return content;
    }
    public void store(T item){
        content = item;
    }
}
```

```java
class BagSubTwo<T,O> extends Bag<T>{
    private O owner;
    public BagSubTwo(){
        this(null, null);
    }
    public BagSubTwo(T item, O owner){
        super();
        store(item);
        setOwner(owner);
    }
    public O getOwner(){
        return owner;
    }
    public void setOwner(O owner){
        this.owner = owner;
    }
}
```
Implementing a generic interface

interface MyContainer<T>
{
    public T retrieve();
    public void store(T item);
}

class MyTrunk<T> implements MyContainer<T>
{
    private T content;
    public MyTrunk()
    {
        store(null);
    }
    public T retrieve()
    {
        return content;
    }
    public void store(T item)
    {
        content = item;
    }
}
Implementing a generic interface

```java
interface MyContainer<T>{
    public T retrieve();
    public void store(T item);
}
```

```java
class MyTrunk<T> implements MyContainer<T> {
    private T content;
    public MyTrunk() {
        store(null);
    }
    public T retrieve() {
        return content;
    }
    public void store(T item) {
        content = item;
    }
}
```

```java
class MyTrunk<T extends Number> implements MyContainer<T> {
}
```
Implementing a generic interface

```java
interface MyContainer<T>
{
    public T retrieve();
    public void store(T item);
}

class MyTrunk<T> implements MyContainer<T> {
    private T content;
    public MyTrunk() {
        store(null);
    }
    public T retrieve() {
        return content;
    }
    public void store(T item) {
        content = item;
    }
}

class MyTrunk<T extends Number> implements MyContainer<T> {
}

class MyStringTrunk implements MyContainer<String> {
}
```
Implementing a generic interface

```java
interface MyContainer<T extends Number>
{
    public T retrieve();
    public void store(T item);
}
```
Implementing a generic interface

```java
interface MyContainer<T extends Number> {
    public T retrieve();
    public void store(T item);
}

class MyTrunk<T extends Number> implements MyContainer<T> {
    public T content;
    public MyTrunk() {
        store(null);
    }
    public T retrieve() {
        return content;
    }
    public void store(T item) {
        content = item;
    }
}
```
Implementing a generic interface

```java
interface MyContainer<T extends Number> {
    public T retrieve();
    public void store(T item);
}

class MyTrunk<T extends Number> implements MyContainer<T> {
    public T content;
    public MyTrunk() {
        store(null);
    }
    public T retrieve() {
        return content;
    }
    public void store(T item) {
        content = item;
    }
}
```
Non-generic implementation

- It is possible to implement a generic interface in a non-generic class.
- This is achieved by providing a specific class for the type parameter in the class definition.
Implementing a generic interface

```java
interface MyContainer<T> {
    public T retrieve();
    public void store(T item);
}

class MyTrunk implements MyContainer<String> {
    private String content;
    public MyTrunk() {
        store(null);
    }
    public String retrieve() {
        return content;
    }
    public void store(String item) {
        content = item;
    }
}
```
All the JCF class have been redefined to be generic classes.

What about earlier programs not written for Java 5 using older JCF classes?

In order to ensure backward compatibility, Java 5 allows a generic class to be created without specifying a type parameter!

Essentially, the compiler assumes that the type is Object -- thus any class can be stored.

If we don’t specify the type, we get a raw type.

Raw types do not have the benefit of type safety and should be avoided.
Arrays and generics

- Arrays should not be used with generics unless you wish to use raw types.
- Instead use the ArrayList or LinkedList class to create a generic array.
- In general, do not use raw types with generics.
- Their use is only for ensuring backward compatibility in legacy code written before Java 5.
Raw Types

Bag bag = new Bag();
bag.store(\texttt{new Integer(5)})
bag.store("test");
bag.store(\texttt{new Book(“Ludlum”)});

Bag rawBag = \texttt{new Bag()};
Bag\langle String\rangle strBag = \texttt{new Bag<\texttt{String}>};
Bag\langle Integer\rangle intBag = \texttt{new Bag<\texttt{Integer}>};

intBag.store(\texttt{new Integer(5)});
rawBag = intBag;
strBag = rawBag;
String str = strBag.retrieve();
Raw Types

```java
Bag bag = new Bag();
bag.store(new Integer(5));
bag.store("test");
bag.store(new Book("Ludlum"));
```

```java
Bag rawBag = new Bag();
Bag<String> strBag = new Bag<String>;
Bag<Integer> intBag = new Bag<Integer>;
intBag.store(new Integer(5));
rawBag = intBag;
strBag = rawBag;
String str = strBag.retrieve();
```

Dangerous, but allowed!
Limitations of Generics

- Cannot create a generic class for
  - Exceptions
  - Anonymous inner classes
    - classes defined within another class, but with no name.
    - E.g. as Action Handlers for GUIs
  - Enum types
    - [http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html](http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html)

- Casting to a generic type
  - generates a warning ("unchecked cast")
  - cast is to the raw type (with no type parameter)
No Inheritance for Generics

- List<Object> is not a super-type of List<Number>
- However, List<Number> is a super-type of ArrayList<Number> !!!!
  This only works if the type parameter is the same.
- Note that arrays behave differently:
  - Object[] is a super-type to Number[]
Lots of subtleties with Generics