# List of Slides

# Java Just in Time

John Latham

February 19, 2019

Chapter 19

# Generic classes

# Chapter aims

- Often wish to have **object** contained within another
  - e.g. may want whole collection of items grouped into one object
  - **list**, **set**, etc..
- Often want collections able to contain *any* kind of object
  - run risk of forgetting what kind of thing are in them
  - like sealing box without labelling it.
- Here introduce idea of applying such labels
  - called **type argument**s.
- Start by exploring problems if don't use such labelling.

Section 2

# Example:

# A pair of any objects

# Aim

*AIM:* To explore potential problems of having a container **object** that can hold **instance**s of any **class**, in particular that we need protection against us erroneously getting the **type** wrong when we extract items from the container. We also introduce the idea of **box**ing an `int` within an `Integer`.

# A pair of any objects

- Introduce example used in next section looking at **generic classes**.
- Have pair of **objects**
  - two items paired together
  - later extracted apart.
- E.g. when desire **method** to **return** two results
  - often use **class variables** or **instance variables** just to receive results
  - or use **array** of length two
    - lack of robustness – e.g. attempt to obtain third item from array – **run time** rather than **compile time** error
- Pair avoids above problems and more elegant.

# The Pair class

```
001: // Two Objects grouped into a pair.
002: public class Pair
003: {
004:   // The two objects.
005:   private final Object first, second;
006:
007:
008:   // Constructor is given the two objects.
009:   public Pair(Object requiredFirst, Object requiredSecond)
010:   {
011:     first = requiredFirst;
012:     second = requiredSecond;
013:   } // Pair
```

# The Pair class

```
016:   // Return the first object.
017:   public Object getFirst()
018:   {
019:     return first;
020:   } // getFirst
021:
022:
023:   // Return the second object.
024:   public Object getSecond()
025:   {
026:     return second;
027:   } // getSecond
028:
029: } // class Pair
```

# The longest argument program

- Contrived but simple example
  - finds longest string in **command line argument**s
  - reports it with its position (counting from one).
  - Finds first occurrence if two or more same greatest length.
- For flexibility/reuse have separate class `LongestString`
  - **class method** to find longest string in **array**.

# The LongestString class

```
001: // Contains a method to find the position of the longest string in an array.
002: public class LongestString
003: {
```

- Our **class method** will **return** `Pair` containing
  - longest string
  - its **array index**.
- But index is **`int`** – **primitive type**
  - `Pair` requires two `Objects`.

# Standard API: `Integer`: as a box for `int`

- `java.lang.Integer` can be used to wrap up **`int`** values as **object**s.
- One **constructor method** given **`int`**
  - creates **instance** wrapping up that number.
  - Known as **box**ing.
- The **instance method** `intValue()` used to retrieve boxed number.
- Allows **`int`** which is **primitive type**
  - to be treated as **object**.

# The LongestString class

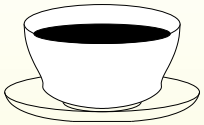
```
004:   // Find the longest string in the given array.
005:   // Return a Pair containing it and its position.
006:   // Throw IllegalArgumentException if array is null or empty.
007:   public static Pair findLongestString(String[] array)
008:                       throws IllegalArgumentException
009:   {
010:     if (array == null || array.length == 0)
011:       throw new IllegalArgumentException("Array must exist and be non-empty");
012:
```

# The LongestString class

```
013:     String longestString = array[0];
014:     int longestIndex = 0;
015:     for (int index = 1; index < array.length; index++)
016:       if (longestString.length() < array[index].length())
017:       {
018:         longestString = array[index];
019:         longestIndex = index;
020:       } // if
021:
022:     return new Pair(longestString, new Integer(longestIndex));
023:   } // findLongestString
024:
025: } // class LongestString
```

# The `LongestString` class

*Coffee time:* What would happen if we swapped the **operand**s of the **conditional or operator** in the first **if statement** above?



*Coffee time:* Our `Pair` **constructor method** expects to be given two `Objects` but we are supplying a `String` and an `Integer`. Is that okay?

# The LongestArgument class

- Observe **cast**s and `intValue()`.

```
001: // Find the longest command line argument and report it and its position.
002: // (Warning: this program does not catch RuntimeExceptions.)
003: public class LongestArgument
004: {
005:   public static void main(String[] args) throws RuntimeException
006:   {
007:     Pair result = LongestString.findLongestString(args);
008:     String longestArg = (String) result.getFirst();
009:     int longestIndex = ((Integer)result.getSecond()).intValue();
010:
011:     System.out.println("A longest argument was `" + longestArg + "'");
012:     System.out.println("of length " + longestArg.length());
013:     System.out.println("found at position " + (longestIndex + 1));
014:   } // main
015:
016: } // class LongestArgument
```

# Trying it

**Console Input / Output**

```
$ java LongestArgument A stitch in time saves nine
A longest argument was ‘stitch’
of length 6
found at position 2
$ java LongestArgument A stitch in time will become very painful
A longest argument was ‘painful’
of length 7
found at position 8
$ _
```

Run

*Coffee time:* What other tests should we perform?

# The LongestArgumentOops class

```
001: // Find the longest command line argument and report it and its position.
002: // (Warning: this program does not catch RuntimeExceptions.)
003: public class LongestArgumentOops
004: {
005:   public static void main(String[] args)
006:   {
007:     Pair result = LongestString.findLongestString(args);
008:     int longestIndex = ((Integer)result.getFirst()).intValue();
009:     String longestArg = (String) result.getSecond();
010:
011:     System.out.println("A longest argument was `" + longestArg + "'");
012:     System.out.println("of length " + longestArg.length());
013:     System.out.println("found at position " + (longestIndex + 1));
014:   } // main
015:
016: } // class LongestArgumentOops
```

# The LongestArgumentOops class

**Console Input / Output**

```
$ javac LongestArgumentOops.java
$ _
```

Run

- The **compiler** believes us
  - we think **type cast**s are okay.
- But it does plant **run time** checks. . . .

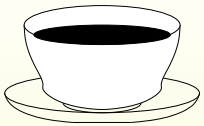
**Console Input / Output**

```
$ java LongestArgumentOops A stitch in time saves nine
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot
 be cast to java.lang.Integer
        at LongestArgumentOops.main(LongestArgumentOops.java:8)
$ _
```

Run

# The LongestArgumentOops class

*Coffee time:* How common do you expect this sort of simple mistake is? Are you happy that the error is only detected at **run time**? What if the error was made in an obscure part of the code that only **execute**s under highly unusual circumstances that were unfortunately not tested for, perhaps during an emergency, such as a sudden close proximity of another aircraft in an auto pilot control program?



# Coursework: A triple

**(Summary only)**

Write a **class** that can store a triple of **object**s, and use it.

Section 3

# Example:
# A generic pair of specified types

# Aim

*AIM:* To introduce the idea of **generic classes**, and show how it can be used to avoid the problems explored in the previous section.

# A generic pair of specified types

- When build **instance** of `Pair`
  - **compiler** knows what **type**s of **object** go into it.
- But when get them out have to tell compiler
  - to **cast** from `Object` to **subclass** we need them to be
  - typically what they were known to be when they went in!
- Type cast checked at **run time**
  - **throw**s `ClassCastException` if got it wrong.
- Would be nice if could allow compiler to already know:
  - what kind of items are in pair?
- In other words, for us to say what *kind* of pair.
- Since Java 5.0 can have **generic class**es.

# Class: generic class

- A **generic class** has one or more **type parameter**s
  - written within <> after name in heading.
- Specific **type**s given as **type argument**s when make **instance**.
- E.g. T1 and T2 are type parameters.

```
public class MyGenericClass<T1, T2>
{
  ... Typical class stuff here,
  ... but using T1 and T2 as though they are types
  ... (in permitted ways).
  private T1 someVariable = ...
  private T2 someOtherVariable = ...
  ...
} // class MyGenericClass
```

# Class: generic class

- Supply specific **type argument** for each **type parameter**
  - e.g. when make instance.

```
MyGenericClass<String, Date> myVariable = new MyGenericClass<String, Date>();
```

- A class is a **type**.
- Intention for generic class is to supply type arguments for type parameters
  - identify **parameterized type**.
- E.g. from `MyGenericClass` can have parameterized types
  - `MyGenericClass<String, Date>`, `MyGenericClass<Integer, String>`, etc.,
  - even `MyGenericClass<String[], Integer>`, etc..

# Class: generic class

- Parameterized type almost behaves as textual copy of generic class
  - but replaced each type parameter
    with corresponding type argument.
- Almost – some restrictions
  - e.g. type arguments must be **reference type**s
    - cannot be **primitive type**s.

# The Pair class

- New version is **generic class** with two **type parameter**s
  - one for **type** of first element of each pair
  - one for second.
- So can have **parameterized type** for any kind of pair.

```
001: // Two Objects grouped into a pair.
002: public class Pair<FirstType, SecondType>
003: {
004:   // The first object.
005:   private final FirstType first;
006:
007:   // The second object.
008:   private final SecondType second;
```

# The Pair class

- The **method parameter**s for **constructor method** not of type `Object`
  - each is appropriate type parameter.

```
011:   // Constructor is given the two objects.
012:   public Pair(FirstType requiredFirst, SecondType requiredSecond)
013:   {
014:     first = requiredFirst;
015:     second = requiredSecond;
016:   } // Pair
```

# The Pair class

- Similarly **return type**s of **accessor method**s.

```
019:   // Return the first object.
020:   public FirstType getFirst()
021:   {
022:     return first;
023:   } // getFirst
024:
025:
026:   // Return the second object.
027:   public SecondType getSecond()
028:   {
029:     return second;
030:   } // getSecond
031:
032: } // class Pair
```

# The LongestString class

- New `findLongestString()` **returns** (a **reference** to) **instance** of **parameterized type** `Pair<String, Integer>`.

```
001: // Contains a method to find the position of the longest string in an array.
002: public class LongestString
003: {
004:   // Find the longest string in the given array.
005:   // Return a Pair containing it and its position.
006:   // Throw IllegalArgumentException if array is null or empty.
007:   public static Pair<String, Integer> findLongestString(String[] array)
008:                                                 throws IllegalArgumentException
009:   {
010:     if (array == null || array.length == 0)
011:       throw new IllegalArgumentException("Array must exist and be non-empty");
012:
```

# The LongestString class

```
013:     String longestString = array[0];
014:     int longestIndex = 0;
015:     for (int index = 1; index < array.length; index++)
016:       if (longestString.length() < array[index].length())
017:       {
018:         longestString = array[index];
019:         longestIndex = index;
020:       } // if
021:
022:     return new Pair<String, Integer>(longestString, new Integer(longestIndex));
023:   } // findLongestString
024:
025: } // class LongestString
```

*Coffee time:* Compare this latest version of `LongestString` with the original in Section 12 on page 12.

# The LongestArgument class

```
001: // Find the longest command line argument and report it and its position.
002: // (Warning: this program does not catch RuntimeExceptions.)
003: public class LongestArgument
004: {
005:   public static void main(String[] args) throws RuntimeException
006:   {
007:     Pair<String, Integer> result = LongestString.findLongestString(args);
008:     String longestArg = result.getFirst();
009:     int longestIndex = result.getSecond().intValue();
010:
011:     System.out.println("A longest argument was '" + longestArg + "'");
012:     System.out.println("of length " + longestArg.length());
013:     System.out.println("found at position " + (longestIndex + 1));
014:   } // main
015:
016: } // class LongestArgument
```

- No need **cast** elements to `String` and `Integer`
  - **compiler** already knows!

# The LongestArgumentOops class

```
001: // Find the longest command line argument and report it and its position.
002: // (Warning: this program does not catch RuntimeExceptions.)
003: public class LongestArgumentOops
004: {
005:   public static void main(String[] args)
006:   {
007:     Pair<Integer, String> result = LongestString.findLongestString(args);
008:     int longestIndex = result.getFirst().intValue();
009:     String longestArg = result.getSecond();
010:
011:     System.out.println("A longest argument was '" + longestArg + "'");
012:     System.out.println("of length " + longestArg.length());
013:     System.out.println("found at position " + (longestIndex + 1));
014:   } // main
015:
016: } // class LongestArgumentOops
```

# The LongestArgumentOops class

*Coffee time:* Do you agree that the above program contains the equivalent error to the one in Section 16 on page 17?

**Console Input / Output**

```
$ javac LongestArgumentOops.java
LongestArgumentOops.java:7: incompatible types
found   : Pair<java.lang.String,java.lang.Integer>
required: Pair<java.lang.Integer,java.lang.String>
    Pair<Integer, String> result = LongestString.findLongestString(args);
                                                                   ^
1 error
$ _
```

Run

# The LongestArgumentOops class

*Coffee time:* While this new power is wonderful to protect against many trivial mistakes, can you think of situations where the accidental swapping of the pair elements would not be detected by the compiler?

*Coffee time:* What do you think would happen if we had not made the changes to the `LongestString` and `LongestArgument` **class**es, but tried to **compile** the original ones from the last section with the generic class version of `Pair`? Try it! Surprised? Can you figure out why it behaves like that?

# Coursework: A generic triple

**(Summary only)**

Write a **generic class** that can store a triple of specific kinds of **object**s, and use it.

Section 4

# Autoboxing and auto-unboxing of primitive values

# Aim

*AIM:* To expose Java's implicit conversion between values of **primitive type**s and **instance**s of the corresponding wrapper **class**es.

# Autoboxing and auto-unboxing of primitive values

- Have seen mechanism for wrapping `int` in `Object`.
- Similar classes for other **primitive type**s.

*Coffee time:* In addition to `Integer`, you have already met *two* other of these wrapper classes, although we have not yet seen them used to wrap up a value. Which two are they?

- Since Java 5.0 have convenience of **autoboxing** / **auto-unboxing**.

# Standard API: `Integer`: as a box for `int`: autoboxing

- Use of `java.lang.Integer` to wrap up **`int`**s is common.
- Since Java 5.0 **compiler** can make use implicit
  - **autoboxing** / **auto-unboxing**.
- Whenever **`int`** given where `Integer` required
  - **`int`** automatically **box**ed.
- Whenever (**reference** to) `Integer` given where **`int`** required
  - `intValue()` automatically used to unbox **`int`**.

# Standard API: `Integer`: as a box for `int`: autoboxing

- E.g.

```
Integer anInteger = new Integer(10);
int anInt = anInteger.intValue() + 1;
System.out.println(anInt);
```

- Following has same effect.

```
Integer anInteger = 10;
int anInt = anInteger + 1;
System.out.println(anInt);
```

- Convenience makes **`int`** and `Integer` **type**s work seamlessly together
  - but most important to remember difference between them:
    - **`int`** is **primitive type**
    - `Integer` is **reference type**.

# Standard API: `Integer`: as a box for `int`: autoboxing

- E.g. **array** of ten **`int`**s (approx) ten times bigger than one **`int`**.
- Array of ten `Integer` objects would hold ten **reference**s,
  - each referring to object storing **`int`** value.

# Autoboxing and auto-unboxing of primitive values

*Coffee time:* Draw a diagram of an **array** of ten `int`s, and another of an array of ten `Integer`s.

*Coffee time:* Do you think autoboxing and auto-unboxing has been applied to the other primitive type wrapper classes? What would be the easiest way to find out?

# Autoboxing and auto-unboxing of primitive values

```
 001: // Contains a method to find the position of the longest string in an array.
 002: public class LongestString
 003: {
...
022:     return new Pair<String, Integer>(longestString, longestIndex);
...
 025: } // class LongestString
```

```
 001: // Find the longest command line argument and report it and its position.
 002: // (Warning: this program does not catch RuntimeExceptions.)
 003: public class LongestArgument
 004: {
...
009:     int longestIndex = result.getSecond();
...
 016: } // class LongestArgument
```

# Coursework: A generic triple, used with autoboxing

**(Summary only)**

Write a **generic class** that can store a triple of specific kinds of **object**s, and use it; this time using **autoboxing** and **auto-unboxing**.

Section 5

# Example:
# A conversation of persons

# Aim

*AIM:* To introduce the idea of a **bound type parameter**, in particular, one that must **extend** some other **type**.

# A conversation of persons

- Enhancement to Notional Lottery – have conversations between `Persons`
  - essentially wrapper around **array** of `Person`
  - with **instance method** `speak()`
    - makes one of the `Persons` speak,
    - repeated calls make each `Person` speak in turn.
- A conversation is kind of *collection* of persons
  - perhaps times when most likely write **generic class** are when implementing collection.
- So we speak of having conversations *of* persons.

# A conversation of persons

- Further, need ability to have particular `Conversations` comprise only persons of particular **subclass**es of `Person`
  - e.g. conversation of `AudienceMembers`,
  - another of `TVHosts`, etc..
- Achieve via **bound type parameter**. . . .

# Class: generic class: bound type parameter

- A **generic class type parameter** may be **bound type parameter**
  - specify certain restrictions on allowed **type argument**s for when **parameterized type** identified.

# Class: generic class: bound type parameter: extends some class

- One kind of restriction for **bound type parameter**:
  - type argument must **extend** some known **class**.
- Follow name of **type parameter** with **reserved word** `extends` and known class.
- The **compiler** checks that type argument is
  - either known class,
  - or **subclass** of it.
- E.g. . . .

# Class: generic class: bound type parameter: extends some class

```
public class ServiceCentre<VehicleType extends Vehicle>
{
  ... Etc., using VehicleType as a type (in permitted ways)
  ... but knowing that it is a Vehicle
  ... and so using some Vehicle methods, etc..

  public void service(VehicleType vehicle)
  {
    if (! vehicle.isRoadworthy())
    {
      ...
    } // if
  } // service

  ...
} // class ServiceCentre
```

# Class: generic class: bound type parameter: extends some class

- Can make `ServiceCentre` **object**s for particular kinds of `Vehicle`.

```
ServiceCentre<Car> garage = new ServiceCentre<Car>();
Car car = new Car(...);
Lorry lorry = new Lorry(...);
garage.service(car);
garage.service(lorry);
garage.service("car");
```

- Last two lines above cause **compile time error**.

# The Conversation class

```
001: // Representation of a group of lottery people talking in turn.
002: public class Conversation<PersonType extends Person>
003: {
```

- So **type argument** given when **parameterized type** identified must be
  - **subclass** of Person,
  - or Person itself.

# The Conversation class

- An **instance** stores (**reference** to) **partially filled array** of `Person` **object**s
  - grown on demand using **array extension**.

```
004:   // Initial size and resize factor.
005:   private static final int INITIAL_ARRAY_SIZE = 2, ARRAY_RESIZE_FACTOR = 2;
006:
007:   // The array, together with the number of Person objects in it.
008:   private Person[] persons = new Person[INITIAL_ARRAY_SIZE];
009:   private int noOfPersons = 0;
```

*Coffee time:* Are you wondering why the array is of type `Person[]` rather than `PersonType[]`? Would that be better?

# The Conversation class

```
012:   // Empty constructor, nothing needs doing.
013:   public Conversation()
014:   {
015:   } // Conversation
```

- addPerson() takes (reference to) object of **type** PersonType, stores in array
  - allowed because PersonType extends Person:
  - it **is a** Person as well as whatever subclass of Person.
- The **compiler** will complain if try to add wrong kind of Person....

# The Conversation class

```
018:   // Add given Person to the Conversation (extend array as required).
019:   public void addPerson(PersonType newPerson)
020:   {
021:     if (noOfPersons == persons.length)
022:     {
023:       Person[] biggerArray = new Person[persons.length * ARRAY_RESIZE_FACTOR];
024:       for (int index = 0; index < persons.length; index++)
025:         biggerArray[index] = persons[index];
026:       persons = biggerArray;
027:     } // if
028:     persons[noOfPersons] = newPerson;
029:     noOfPersons++;
030:   } // addPerson
```

# The Conversation class

*Coffee time:* Are you getting tired of seeing code that copies from one array to another? Take a look in the **API** documentation for the `System` class to find something that might be of interest to you.

```
033:   // Return the number of people in the conversation.
034:   public int getSize()
035:   {
036:     return noOfPersons;
037:   } // getSize
```

# The Conversation class

```
040:   // Used to keep track of whose turn it is to speak.
041:   private int nextToSpeak = 0;
042:
043:
044:   // Make the next person speak and update who is next after that.
045:   public void speak()
046:   {
047:     if (noOfPersons > 0)
048:     {
049:       persons[nextToSpeak].speak();
050:       nextToSpeak = (nextToSpeak + 1) % noOfPersons;
051:     } // if
052:   } // speak
```

# The Conversation class

```
055:   // Mainly for testing.
056:   @Override
057:   public String toString()
058:   {
059:     String result = noOfPersons == 0 ? "" : "" + persons[0];
060:     for (int index = 1; index < noOfPersons; index++)
061:       result += String.format("%n%s", persons[index]);
062:     return result;
063:   } // toString
064:
065: } // class Conversation
```

# The TestConversation class

```
001: // Create conversations of persons and make them speak.
002: public class TestConversation
003: {
004:   public static void main(String[] args)
005:   {
```

- Conversation in which all persons *must* be AudienceMembers:
  - **compiler** checks do not add wrong kind of Person.

```
006:     // A conversation of AudienceMembers.
007:     Conversation<AudienceMember> audienceChat
008:       = new Conversation<AudienceMember>();
009:     audienceChat.addPerson(new AudienceMember("AM 1"));
010:     audienceChat.addPerson(new AudienceMember("AM 2"));
011:     audienceChat.addPerson(new AudienceMember("AM 3"));
```

# The TestConversation class

```
012:     System.out.printf("%s%n%n", audienceChat);
013:     for (int count = 1; count <= audienceChat.getSize(); count++)
014:     {
015:       audienceChat.speak();
016:       System.out.printf("%s%n%n", audienceChat);
017:     } // for
```

*Coffee time:* How can we have a conversation of any kind of person?

# The TestConversation class

```
019:     // A conversation of any kind of person.
020:     Conversation<Person> anyChat = new Conversation<Person>();
021:     anyChat.addPerson(new TVHost("TVH 1"));
022:     anyChat.addPerson(new AudienceMember("AM 4"));
023:     System.out.printf("%s%n%n", anyChat);
024:     for (int count = 1; count <= anyChat.getSize(); count++)
025:     {
026:       anyChat.speak();
027:       System.out.printf("%s%n%n", anyChat);
028:     } // for
029:   } // main
030:
031: } // class TestConversation
```

# Trying it

## Console Input / Output

```
$ java TestConversation
 (Output shown using multiple columns to save space.)
Audience Member AM 1 true I am AM 1
Audience Member AM 2 true I am AM 2
Audience Member AM 3 true I am AM 3

Audience Member AM 1 true Oooooh!
Audience Member AM 2 true I am AM 2
Audience Member AM 3 true I am AM 3

Audience Member AM 1 true Oooooh!
Audience Member AM 2 true Oooooh!
Audience Member AM 3 true I am AM 3

Audience Member AM 1 true Oooooh!
$ _
```

```
Audience Member AM 2 true Oooooh!
Audience Member AM 3 true Oooooh!

TV Host TVH 1 true I am TVH 1
Audience Member AM 4 true I am AM 4

TV Host TVH 1 true Welcome, suckers!
Audience Member AM 4 true I am AM 4

TV Host TVH 1 true Welcome, suckers!
Audience Member AM 4 true Oooooh!
```

Run

# The TestConversationOops class

```
001: // Create conversations of people and make them speak.
002: public class TestConversationOops
003: {
004:   public static void main(String[] args)
005:   {
006:     // A conversation of AudienceMembers.
007:     Conversation<AudienceMember> audienceChat
008:       = new Conversation<AudienceMember>();
009:     audienceChat.addPerson(new AudienceMember("AM 1"));
010:     audienceChat.addPerson(new TVHost("TVH 1"));
011:     System.out.printf("%s%n%n", audienceChat);
012:     for (int count = 1; count <= audienceChat.getSize(); count++)
013:     {
014:       audienceChat.speak();
015:       System.out.printf("%s%n%n", audienceChat);
016:     } // for
017:   } // main
018:
019: } // class TestConversationOops
```

# The TestConversationOops class

**Console Input / Output**

```
$ javac TestConversationOops.java
TestConversationOops.java:10: addPerson(AudienceMember) in Conversation<Audience
Member> cannot be applied to (TVHost)
    audienceChat.addPerson(new TVHost("TVH 1"));
                 ^
1 error
$ _
```

Run

# The TestConversationOops class

*Coffee time:* Recall the full `Person` hierarchy from Section **??** on page **??**. How could we have a `Conversation` in which all the persons must be `MoodyPersons`, but can be any kind of moody person?

*Coffee time:* Recall that within the `Conversation` **class**, we had an **array** of **type** `Person[]`, in which only `PersonType` **object**s were stored. It would have been nicer to declare the array as `PersonType[]`. So, why didn't we? Try it to find out!

# Coursework: A moody group

**(Summary only)**

Write a **generic class** that can store a collection of a particular kind of `MoodyPerson` **object**s, from the Notional Lottery example, and make them all happy or unhappy at the same time.

Section 6

# What we cannot do with type parameters

# Aim

*AIM:* To briefly explore some of the things we might like to do with **type parameter**s but cannot.

# Class: generic class: where type parameters cannot be used

- Each **type parameter** of **generic class** may be treated as **type** within that class
  - but certain restrictions, in two categories.
- First, meaning of type parameters:
  - **type argument** is supplied for each parameter to identify **parameterized type**
  - ready for **instance**s to be made.
  - Type arguments only mean anything in context of creating instances
  - make no sense in **static context** of generic class
    - (which is not part of the type).
  - We cannot refer to type parameters in `static` parts
    - **class variable** and **class method** declarations.

# Class: generic class: where type parameters cannot be used

- Second set of restrictions about way Java implements generic classes.
  - Cannot create any **instance**s of type parameter
  - nor **array**s with **array element**s of that type.
  - Generic features is entirely **compile time** artifact
    - enables **compiler** undertake more type checking.
  - At **run time**, **virtual machine** has no knowledge of type parameters
    - so cannot *create* instances of them.

# What we cannot do with type parameters

- Cannot have this – pity?

```
001: // Create instances of ObjectType, and count them.
002: public class CountingFactory<ObjectType>
003: {
004:   // The number of instances made so far.
005:   private int constructionCountSoFar = 0;
006:
007:
008:   // Empty constructor, nothing needs doing.
009:   public CountingFactory()
010:   {
011:   } // CountingFactory
012:
013:
```

# What we cannot do with type parameters

```
014:   // Return the number of objects that have been made up to now.
015:   public int getConstructionCount()
016:   {
017:     return constructionCountSoFar;
018:   } // getConstructionCount
019:
020:
021:   // Create an ObjectType and count it.
022:   public ObjectType newObject()
023:   {
024:     constructionCountSoFar++;
025:     return new ObjectType();
026:   } // newObject
027:
028: } // class CountingFactory
```

# Trying it

**Console Input / Output**

```
$ javac CountingFactory.java
CountingFactory.java:25: unexpected type
found   : type parameter ObjectType
required: class
    return new ObjectType();
               ^
1 error
$ _
```

Run

Section 7

# Using a generic class without type parameters

# Aim

*AIM:* To briefly explore what happens when we use a **generic class** without **type parameter**s.

# Class: generic class: used as a raw type

- A **generic class** is still a **class**
  - and hence a **type**
  - can be used directly to make **instance**s without supplying **type argument**s.
- Due to legacy issues:
  - generic classes added in Java 5.0
  - **type parameter**s added to many standard **API** classes
  - already existed millions of Java programs using those classes
  - unacceptable for all to suddenly stop working!

# Class: generic class: used as a raw type

- Type of generic class without type parameters called its **raw type**.
- When use raw types **compiler** assumes best known actual type for each type parameter
  - gives warnings about types being `unchecked`.
- But makes **byte code** anyway.
- Programmers encouraged to use generic classes properly for new code
  - and gradually change legacy code.
- Best known type assumed by compiler for type parameter which **extend**s some concrete type is that concrete type
  - for ones that do not it is `java.lang.Object`.

# Using a generic class without type parameters

```
001: // Create conversations of people and make them speak.
002: public class TestConversationOops
003: {
004:   public static void main(String[] args)
005:   {
006:     // A conversation of AudienceMembers.
007:     Conversation audienceChat = new Conversation();
008:     audienceChat.addPerson(new AudienceMember("AM 1"));
009:     audienceChat.addPerson(new TVHost("TVH 1"));
010:     System.out.printf("%s%n%n", audienceChat);
011:     for (int count = 1; count <= audienceChat.getSize(); count++)
012:     {
013:       audienceChat.speak();
014:       System.out.printf("%s%n%n", audienceChat);
015:     } // for
016:   } // main
017:
018: } // class TestConversationOops
```

# Trying it

**Console Input / Output**

```
$ javac TestConversationOops.java
Note: TestConversationOops.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
$ _
```

Run

- The **compiler** does not give details of warnings
  - but can ask for details with `-Xlint:unchecked` compiler option...

# Trying it

**Console Input / Output**

```
$ javac -Xlint:unchecked TestConversationOops.java
TestConversationOops.java:8: warning: [unchecked] unchecked call to addPerson(Pe
rsonType) as a member of the raw type Conversation
    audienceChat.addPerson(new AudienceMember("AM 1"));
                           ^

TestConversationOops.java:9: warning: [unchecked] unchecked call to addPerson(Pe
rsonType) as a member of the raw type Conversation
    audienceChat.addPerson(new TVHost("TVH 1"));
                           ^

2 warnings
$ _
```

Run

- Should not write new code that generates warnings like this.

# Trying it

- Most worryingly our erroneous program **run**s without errors!

**Console Input / Output**

```
$ java TestConversationOops
Audience Member AM 1 true I am AM 1
TV Host TVH 1 true I am TVH 1

Audience Member AM 1 true Oooooh!
TV Host TVH 1 true I am TVH 1

Audience Member AM 1 true Oooooh!
TV Host TVH 1 true Welcome, suckers!

$ _
```

Run

# The TestConversationMajorOops class

```
001: // Create conversations of people and make them speak.
002: public class TestConversationMajorOops
003: {
004:   public static void main(String[] args)
005:   {
006:     // A conversation of AudienceMembers.
007:     Conversation audienceChat = new Conversation();
008:     audienceChat.addPerson("AM 1");
009:     System.out.printf("%s%n%n", audienceChat);
010:     for (int count = 1; count <= audienceChat.getSize(); count++)
011:     {
012:       audienceChat.speak();
013:       System.out.printf("%s%n%n", audienceChat);
014:     } // for
015:   } // main
016:
017: } // class TestConversationMajorOops
```

# The TestConversationMajorOops class

- At least get **compile time error**
  if try to add **object** which is not a Person.

**Console Input / Output**

```
$ javac TestConversationMajorOops.java
TestConversationMajorOops.java:8: addPerson(Person) in Conversation cannot be ap
plied to (java.lang.String)
    audienceChat.addPerson("AM 1");
                 ^
1 error
$ _
```

Run

# Concepts covered in this chapter

- Each book chapter ends with a list of concepts covered in it.
- Each concept has with it
  - a self-test question,
  - and a page reference to where it was covered.
- Please use these to check your understanding before we start the next chapter.