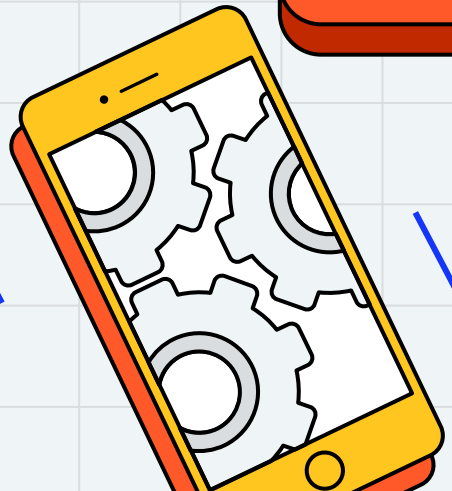
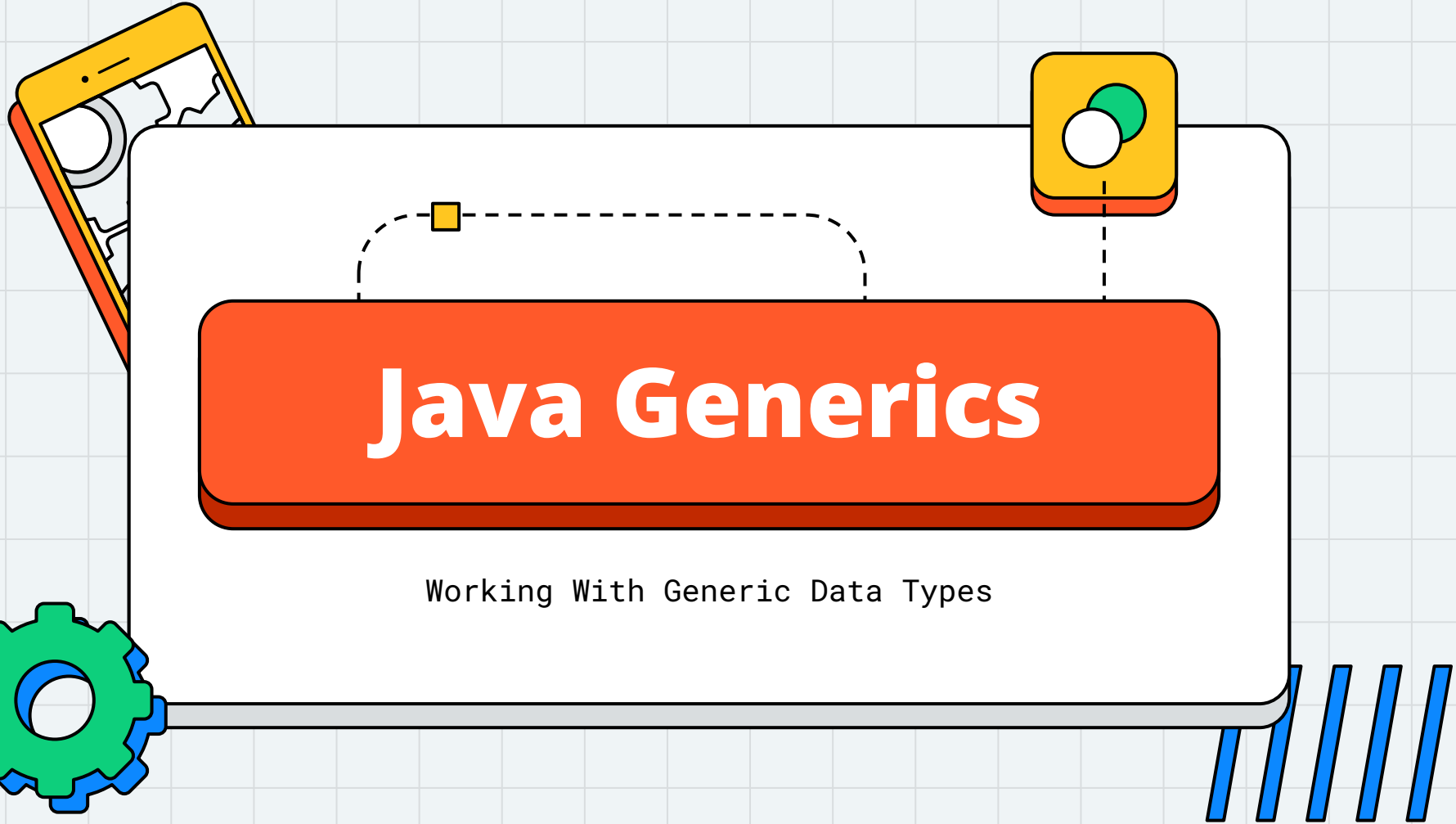
Application Programming



Hend Alkittawi


Java Generics
Working With Generic Data Types

INTRODUCTION

- An arraylist of Strings can be created **with generics**, as in

```
ArrayList<String> list = new ArrayList<String>();
list.add("hello"); // Add a String object to the list
```

- In order to retrieve a String object from the list, we do not need to cast it to a String type

```
String s = list.get(0);
```

- When an arraylist of String objects is created with generics, the following statements are **not valid** statements

```
// Adding an Integer to the String list produces a compilation error
list.add(0);
```

INTRODUCTION

- An arraylist of Strings can be created **without generics**, as in

```
ArrayList list = new ArrayList();
list.add("hello"); // Add a String object to the list
```

- In order to retrieve a String object from the list, we must cast it to a String type

```
String s = (String) list.get(0);
```

- When an arraylist of String objects is created without generics, the following statements are valid statements

```
list.add(0);        // Add an Integer to the list
Integer s = (Integer) list.get(1);
```

JAVA GENERICS

- In a nutshell, generics enable **types** (classes and interfaces) to be **parameters** when defining classes, interfaces and methods
- Advantages to using generics in your code
  - Enables programmers to implement **generic algorithms**
    - focus is on creating more elegant algorithms, rather than syntax
    - still generates clean, type-safe, customizable code
  - Eliminates **casting**
  - **Stronger type checks** are performed at compile time
    - if a problem exists, it’s better to find it at compile time than at run time!

GENERIC METHODS

- Creating generic methods enables code reuse and simplifies your code.
- For example:
  - A method which takes in an array of numbers, and returns a random element in the array:

```
public Integer getRandomElement(Integer[] array) {
    Random random = new Random();
    int index = random.nextInt(array.length);
    Integer result = array[index];
    return result;
}
```

  - A method which takes in an array of Strings, and returns a random element in the array:

```
public String getRandomElement(String[] array) {
        Random random = new Random();
        int index = random.nextInt(array.length);
        String result = array[index];
        return result;
}
```

GENERIC METHODS

- From the previous example, a generic method that enables code reuse and simplifies the code can be created

```
public <T> T getRandomElement(T[] array) {
        Random random = new Random();
        int index = random.nextInt(array.length);
        T result = array[index];
        return result;
}
```

- <T> indicates that the method will be a generic method
- T indicates the type of the generic variable in the method
  - defined when the method is called!

GENERIC METHODS

- A generic method that enables code reuse and simplifies your code!

```
public <T> T getRandomElement(T[] array) {
        Random random = new Random();
        int index = random.nextInt(array.length);
        T result = array[index];
        return result;
}
```

- We can call this method as follows

```
Integer[] intArray = { 1, 3, 5, 7, 9, 0, 2, 4, 6, 8 };
String[]  stringArray = { "xx", "yy", "zz", "aa", "bb", "cc" };
Integer rint = getRandomElement(intArray);
String rstring = getRandomElement(stringArray);
```

GENERIC CLASSES

- A **generic class** is a class that is parameterized over types.
- One of the key concepts of Java generics is that only the compiler processes the generic parameters.
- The Java **compiler** uses generics to ensure **type-safety**.
- There is no generic type-checking in the runtime code.
- Essentially, generic classes avoid the same issues faced in our previous example for generic methods.
  - Think about a generic Stack class!

```java
/**
* A PairOfIntegers object stores a pair
* of Integers.
*
* @author Tom Bylander
*/
public class PairOfIntegers {

    private Integer first, second;

    public PairOfIntegers(Integer
            integer1, Integer integer2){
        first = integer1;
        second = integer2;
    }

    public Integer getFirst() {
        return first;
    }

    public Integer getSecond() {
        return second;
    }
}
```

```java
/**
* A PairOfStrings object stores a pair
of strings.
*
* @author Tom Bylander
*/
public class PairOfStrings {

    private String first, second;

    public PairOfStrings(String string1,
                         String string2){
        first = string1;
        second = string2;
    }

    public String getFirst() {
        return first;
    }

    public String getSecond() {
        return second;
    }
}
```

```java
/**
* A PairOfIntegerAndDouble object stores
an Integer and a Double.
*
* @author Tom Bylander
*/
 public class PairOfIntegerAndDouble {
    private Integer first;
    private Double second;


    public PairOfIntegerAndDouble(
          Integer first, Double second){
        this.first = first;
        this.second = second;
    }


    public Integer getFirst() {
        return first;
    }

    public Double getSecond() {
        return second;
    }
}
```

```java
/**
* A PairOfObjects object stores a pair
* of objects.
*
* @author Tom Bylander
*/


public class PairOfObjects {



    private Object first, second;

    public PairOfObjects(Object object1,
                         Object object2){
        first = object1;
       second = object2;
    }

    public Object getFirst() {
        return first;
    }

    public Object getSecond() {
        return second;
    }
}
```

```java
PairOfObjects pair7 =
               new PairOfObjects(123, 0.456);
```

```java
/**
* A PairOfSameType object stores a pair * of
objects generic type T.
* T is specified when
* declaring the PairOfSameType variable.
*
* @author Tom Bylander
*/

public class PairOfSameType<T> {

    // Note the use of the generic
    // parameter T.
    private T first, second;
    Private ArrayList<T> myList;

    public PairOfSameType(T object1,
                                T object2){
        first = object1;
        second = object2;
    }

    public T getFirst() {
        return first;
    }

    public T getSecond() {
        return second;
    }
}
```

```java
PairOfSameType<Integer> pair5 =
   new PairOfSameType<Integer>(123,456);
```

```java
/**
* A PairOfDifferentTypes object stores a *
pair of objects.
* The type parameter S is for the first *
object.
* The type parameter T is for the second *
object.
* @author Tom Bylander
*/
public class PairOfDifferentTypes<S,T> {
    // Note the use of S and T.
    private S first;
    private T second;


    public PairOfDifferentTypes(
                 S object1, T object2){
        first = object1;
        second = object2;
    }

    public S getFirst() {
        return first;
    }

    public T getSecond() {
        return second;
    }
}
```

```java
PairOfDifferentTypes<Integer,Double> pair9 =
     new PairOfDifferentTypes<Integer,Double>
                                (123, 0.456);
```

THANK YOU!

DO YOU HAVE ANY QUESTIONS?

hend.alkittawi@utsa.edu

By Appointment

Online